AFIT/GE/ENG/94M-04

EFFECTIVE PARALLEL ALGORITHM ANIMATION

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Engineering)

Paul William Chase, B.E.

FLTLT, RAAF

March, 1994

Acknowledgements

This thesis effort required the support and understanding of a number people. In particular my wife Sue deserves as much of the credit for this thesis as I do, since her sacrifices and loving support made it possible. Of course Dr. Lamont also deserves credit for providing the motivation and guidance by continually providing interesting challenges.

To the friends I made during the course, in particular Vince Droddy and many others, thanks for the memories, you made my time in the USA great.

Finally, I would like to thank the other AFIT lecturers who re-kindled my interest in engineering, and for a time, saved me from a mundane life of administration.

Fltlt Paul William Chase

Table of Contents

Page

Page

Page

List of Figures

List of Tables

AFIT/GCE/ENG/94M-04

Abstract

Algorithm animation can provide significant insight into the execution of serial and parallel programs. In particular, research into its use in program debugging, optimization, and pedagogical observation has been extensive. Systems such as Paragraph, Pablo, and the Air Force Institute of Technology's (AFIT) Algorithm Animation Research Facility (AAARF) represent significant animation environments for parallel systems. Most systems provide generic animations based on message passing activity which can be further enhanced by using application specific displays. The goal of this investigation is to: 1) create both general and application specific displays that support visualization of parallel algorithms, and 2) consider appropriate visualization system enhancements for massively parallel architectures. Thus, this research focuses in part on developing animation support for current AFIT parallel processing research projects. Analysis, design, and implementation of application specific displays for discrete event simulation, mission routing, and evolutionary algorithms are based on abstract representations of parallel algorithm behavior. The effort also builds extensions to the AAARF system and examines direction for further research. An innovative adaptable application-specific animation construction environment has been designed and implemented. The advantages of this revised type of animation environment are clearly demonstrated with its successful application to genetic algorithm visualization. The environment provides a set of composible graphics objects and a flexible event mapping and transformation system that allows development of arbitrary animation formats. The system also reduces operational complexity via a simplified event format. The ability to compose arbitrary animations, without coding, is currently not integrated with other similar systems. The resulting visualization system is inherently portable between architectures, easily extensible to meet specific user animation requirements, and successfully deals with scalability problems associated with massively parallel processing systems.

EFFECTIVE PARALLEL ALGORITHM ANIMATION

I. *Effective Parallel Algorithm Animation*

1.1 Introduction

Parallel processing provides an opportunity to gain virtually unlimited computational p
by decomposing a task into subtasks that can be executed on a number of processing elements. The scalability of parallel processing systems currently provides the only method of achieving the computational power required to solve a vast array of complex problems. From a system perspective, scalability is the inherent effectiveness of adding additional processing elements to the computational platform. If a system is scalable, the additional processing elements are capable of reducing the total program execution time for scalable algorithm implementation. From an algorithm perspective, a scalable algorithm is one that is capable of effectively utilizing the additional processing elements.

To utilize the computational power of a parallel processing system, a problem must be decomposed to make use of the available processors. If this decomposition is not effective, the parallel implementation may, in fact, perform worse than the original serial program. Current design techniques provide some guidance for decomposing a given problem. However, the complex execution patterns of many parallel programs lead to the need to analyze their actual execution (29, 62, 28). This empirical approach is often necessary due to the analysis and computation time required to determine a theoretical decomposition and task schedule, or simply the non-determinism inherent in the underlying algorithm. In most realistic applications, task scheduling and decomposition requires non-polynomial time complexity algorithms (22). While theoretical methods exist (10, 46) current techniques provide only limited modeling of the target system and usually produce suboptimal solutions. As a result the task schedules and decompositions produced using these techniques often lead to undesired real-world behavior. If the behavior of the required algorithm is non-deterministic, theoretical approaches at best provide only approximate solutions. These difficulties lead to the lack of general methods for predicting application or system behavior on massively parallel systems (47, 10). In addition, there are few tools that allow for post-execution analysis.

Program animation is the process of graphically representing the state changes which occur during the execution of the control structure (algorithm) of the computer program. When the animations are extended to include high level algorithm events (84:41), the logical question of why a program behaved in a particular manner can be addressed.

Algorithm animation is vital in the development of new parallel software and the optimization of existing software. Without its use during execution analysis, real-world algorithm behavior is often hidden from the user. Moreover it is an effective aid in teaching algorithm design and for understanding the execution behavior of existing algorithms.

1.2 Background

AFIT has been extensively involved in research into algorithm animation. Research by Fife (16), Williams (84), Lack (43), and Wright (86) produced the current AFIT Algorithm Animation Research Facility (AAARF). This system is a general purpose visualization tool for animation of serial and parallel algorithms. AAARF has been used to examine the viability of real time data collection , low level system visualizations, and high level algorithm displays (84) for parallel algorithm observation.

AAARF was designed and implemented as an animation system for serial processes by Fife (16) in 1988-1989. It was extended by Williams (84) in 1989-1991 to include animations for parallel performance analysis and parallel algorithms on the Intel iPSC/2 Hypercube. Lack (43) further extended AAARF during 1990-1991 by adding an expert system advisor and additional iPSC/2 animations. The latest revision by Wright (86) in 1991-1992, ported the system from the SunView (86:49) programming environment to Xview (86:41).

Previous AAARF research and development has produced a functional tool that provides the basis from which to examine algorithm animation. A number of sample applications have been instrumented and a limited number of application specific displays developed. The applications instrumented include : A^* based set covering program (84:5-2), a collection of parallel sorting routines (84:A-1), and a ring communication program (86:B-1). The two application specific displays developed for parallel program support include a bar graph view for sorting routines and a tree building animation for search algorithms (84:A-1).

1.3 Outline of Research Problems

This research investigates how to enhance and extend the current algorithm animations capabilities of the AAARF system, the aim being to directly support other AFIT research projects with informative animations. Currently algorithm research is significantly restricted by the availability of tools for execution analysis. In particular, current parallel execution analysis tools:

- Lack the animation scalability necessary to meet the demands of massively parallel systems.
- Lack integration of algorithm data animations.
- Impose significant perturbations on the programs executions.
- Produce voluminous trace data that limits the period of observation.

The additional animation support is directed towards algorithm understanding and analysis capabilities, while reducing the limitations outlined above. With increasing usage of massively parallel processing systems, visualization support must provide more than just informative animations. For additional animations to be effective they must be scalable, extensible, portable, and most importantly reveal aspects of the program's execution previously not visible to the user.

The system must be scalable to allow depiction of applications executing on massively parallel computer systems; and extensible to allow creation of new depictions, for execution behavior not previously encountered. The displays must promote portability to allow comparison of execution results on various parallel systems. Most importantly, the displays must provide information about the algorithm under examination that is of benefit to the user. This benefit may be providing educational understanding of execution behavior, or assistance in development and enhancement of the algorithm.

The central problem addressed by this research is the development of visualization tools that meet these requirements.

1.4 Objectives

The problems outlined in the previous section require that a number of current animation research questions must be investigated.

Providing support for current AFIT research projects demands that specific display classes be examined for each application area. The generic data provided by the standard AAARF displays is inadequate for in-depth algorithm research, which requires greater knowledge of program execution parameters. For example, current generic animations provide no information relating to program data structures and control flow within a processing node. The first objective is to investigate the applicability and capabilities of this animation approach. Opportunities also exist for the enhancement of the generic message based animation currently provided by AAARF. The second objective is thus, to examine additional generic displays and their applicability to enhancement of specific application animation objectives.

AFIT's parallel research activity is now focusing on the Intel iPSC/860 for development work and AAARF must be capable of supporting this platform. To ensure that the animation produced provides accurate data, we must examine and quantify the effects of execution tracing. The more advanced Intel Paragon (39) system poses additional animation requirements. This research also aims to provide clear direction for further AAARF development that supports the Paragon system.

The current AAARF system poorly addresses the problems of scalability and portability. AAARF is currently designed around the concept of a fixed architecture with a limited number of processing nodes (≤ 16). To address these problems the development of animations are system independent, in terms of the number of processing nodes and interconnection arrangements which are required. To address portability requirements, these animation displays must also process event traces that can be generated on any parallel system.

Extensibility is a major objective of this research. The requirement for users to development software when generating application specific displays must be eliminated. It is, for example, unacceptable to expect the user of the animation tool to develop Xwindow's graphics code to produce the required animation formats.

Thus, the basic objective of providing animation support for parallel algorithm research can be divided into sub-goals as follows.

- The development of application specific animations that support the major AFIT parallel research projects:
  - Parallel Discrete Event Simulation (PDES) (5).
  - Parallel Genetic Algorithms (52, 6, 15, 73).
  - Multi-Criteria Mission Routing (14).
- The instrumentation and animation of other algorithm types not currently represented in the AAARF program library.
- Porting of the data collection software (PRASE (16)) to iPSC/860. This objective satisfies the requirement to support other processing platforms used by AFIT researchers.

- Investigation of animation overhead and enhancement of the data collection facility. This activity provides information required to appropriately apply algorithm animation.
- Development of animation formats that are inherently scalable and portable.
- Development of techniques for generation of these displays that do not require coding by the user. Achieving this objective reduces/eliminates the impact of problems inherent in current animation systems.

The fulfillment of these objectives provide enhancements to the AAARF animation system that resolve the problems discussed in Section 1.3.

1.5 Summary of Current Knowledge in Parallel Algorithm Visualization

The field of parallel program visualization has developed over the last decade to a point where complete systems are available for use during the software development cycle (Paragraph (29), Pablo (62) AAARF (84), and AIMS (57)). The Pablo system developed by Reed (62) contains some of the most advanced design features with self defining event tracing and user configurable display formats. While the tools currently available provide significant capabilities, further improvement is required before these systems can be utilized on more complex programs. The current problem areas of parallel algorithm visualization can be divided into three main research topics:

- Reduction of trace data volume, minimizing tracing overhead. and trace format standardization.
- Development of scalable animations of process/machine level events that are processing system independent.
- Development of scalable abstract animations of high level algorithm events.

These are discussed in more detail in Chapter II.

1.5.1 Reduction of Trace Data Volume, and Instrumentation Overhead and Trace Format Standardization. Development is yet to make significant inroads into the problem of trace data volume. While a number of techniques have been tried, none have achieved an order of magnitude reduction in the amount of trace data required. Current research includes the use of trace predicates that evaluate the data as it is collected (17). This scheme discards information that is not required for a particular animation sequence. The use of real time data collection and analysis has also been examined (84). To be truly useful, the data collection system must also limit its effect on the algorithm under examination (artifact). To this end, some preliminary research into quantifying these effects has been reported (71, 51).

Only recently have efforts been made to produce a standardized trace data format (12, 62). While trace standardization plays an important part in the development of animation systems, standardization has currently failed to gain wide acceptance. This is mainly a result of the continuing effort to reduce the trace data volume and provide event formats for specific processing platforms. The most promising activity in this area is the development of self defining trace format (62:51). This system use a meta-file format that can be interpreted from the event descriptions provided in the trace data. This system is likely to develop as a standard since it has now been incorporated into the OSF operating system on the Intel Paragon (39).

1.5.2 Animation of Process/Machine Level Events. Process level animations are the most widely studied implementations of program animation. Currently a number of systems exist (49, 29, 60, 3, 24, 62) that can produce a variety of animations. These systems have been extensively utilized in analysis of actual application programs (19, 28, 62). Many current animation systems are effectively limited to displaying up to 512 processing elements. Systems that provide animation support for larger numbers of nodes, Pablo (62) for example, utilize equivalence classes to allow mapping of processor activity to a reduced number of display objects. The equivalence classes result from the regularity inherent in many parallel algorithm implementations. In these situations, groups of processors execute similar/identical code segments and thus classes of processor activity can be defined by the user. Research continues to examine new display formats that can cope with massive parallel computer systems containing thousands of processing elements (62. 67).

1.5.3 Animation of High Level Parallel Algorithm Events. High level algorithm animation has been applied in various forms to serial processing systems (2, 16). Only recently have these techniques been applied to parallel programs. A number of systems include algorithm animation capabilities, for example Paragraph (29), VISTA (67), Pablo (62) , and AAARF (84). Currently systems such as VISTA, Paragraph, AIMS (57) and others focus on generation of specific animations for different algorithm classes. The Pablo (62) research extends this principle by providing a development environment for users to create particular instances of an animation class.

In this research we examine both the low and high level animations and implement a number of new animations in both areas which have been incorporated into the AAARF animation system. From this research, the characteristics of future parallel algorithm animation systems are defined. The aim of this activity is to initiate the development of a new generation of the AAARF system.

1.6 Assumptions

The main assumption made in this research is that all work undertaken is based on the current implementation of AAARF (86). While other systems are available, AAARF has the advantage of already being integrated into a number of AFIT research projects. Alternative systems are discussed in Chapter II. Currently, AAARF is primarily used as an analysis and debugging tool for programs developed by AFIT. It is assumed that the local use of the system remains AAARF's primary focus.

1.7 Scope

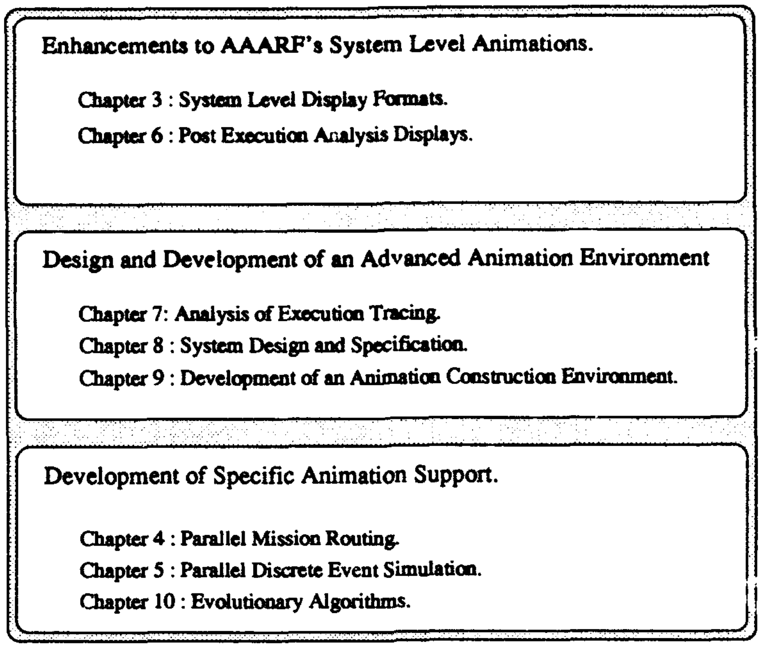
The scope of this research is confined to the list of tasks given in the objective. Within these bounds, correction of existing software bugs and the establishment of version control was undertaken. The animations focus exclusively on programs implemented on message passing computer architectures. Different algorithms are instrumented to test the results of developed techniques, but algorithm research is not part of this study. This research results in additions to the current AAARF system.

1.8 Overview of Research Activity

To achieve these research objectives a number of developmental and experimental activities were undertaken.

Extensive experimentation with the current AAARF animation, and other animation research tools, Paragraph (29) and Pablo (62), lead to enhancement of the generic AAARF animations. In particular Chapter III investigates additional run-time displays. These displays emphasize aspects of the program's performance not previously depicted by AAARF. Based on the application of these animations to a range of programs which are listed in Section 3.2, a new post analysis tool was developed. This interactive tool provides post execution analysis data in a graphical format that complements the run-time depictions. The development and implementation of this tool is discussed in Chapter VII. This work provided the foundation for the development of new animations that directly support other AFIT research projects.

Experimentation with application specific displays for different algorithm classes is presented in Chapters IV & V . In these chapters animation support for mission routing (14) and parallel discrete event simulation (5, 83) is discussed. The animations developed to support these research areas represent the first such attempts to provide animations for these algorithm classes. In particular, the animations for parallel discrete event simulation, show that useful algorithm animations



Enhancements to AAARF's System Level Animations.

Chapter 3 : System Level Display Formats.
Chapter 6 : Post Execution Analysis Displays.

Design and Development of an Advanced Animation Environment

Chapter 7: Analysis of Execution Tracing.
Chapter 8 : System Design and Specification.
Chapter 9 : Development of an Animation Construction Environment.

Development of Specific Animation Support.

Chapter 4 : Parallel Mission Routing.
Chapter 5 : Parallel Discrete Event Simulation.
Chapter 10 : Evolutionary Algorithms.

Figure 1. Thesis Outline

can be developed for programs that exhibit high degrees of non-determinism. These animations were developed using the existing AAARF code based animation construction techniques.

To meet our objective of providing extensible animation formats that are inherently scalable and portable, a new direction was taken for the AAARF system. A detailed analysis of event tracing and animation requirements (see Chapters VII & VIII) lead to specification of a new animation environment. From this specification, an animation generation environment was developed. This environment allows the construction of arbitrary animations without coding. The resultant animations can be developed independently of the number of processing nodes and system architecture. Within this framework the equivalence classes, often present in parallel program behavior, can be used to produce displays capable of scaling to massively parallel systems. The adaptive display environment is discussed in Chapter IX.

To demonstrate the advantages of the adaptive animation environment, it was utilized to generate animation support for parallel genetic algorithm research, discussed in Section X. The displays generated were flexible and easily tailored to satisfy specific user analysis criteria. User definable display format files reduce display generation times to less than two hours, where previously 30 to 40 hours could be expected for displays of equal complexity. These times are based on experience gained during this research. The resultant displays can be developed independently of the number of nodes, trace event format, and system architecture. This final experimentation demonstrated that the adaptive animation environment truly met our goals of scalability, extensibility, and portability.

1.9 Summary

Algorithm animation provides significant insight into the execution of parallel programs. While the use of generic animations can provide useful information, this can be further enhanced by using application specific displays. This investigation developed animations that are scalable, extensible, portable, and directly assist in analysis, via animation, of current AFIT parallel research projects. Figure 1 provides a graphical outline of the remainder of this report, and shows the focus of each chapter. The next chapter contains the requirements analysis for animation systems and a review of current program visualization and parallel instrumentation packages.

II. Parallel Algorithm Animation: Summary of Requirements, Design, and Implementation Issues

2.1 Introduction

The field of parallel program visualization has developed over the last decade to a point where extensive systems can be used during the software development cycle. The majority of visualization systems typically provide various time-based displays of inter-processor communications, event sequences, and processor activity. The inter-processor communication events represent the sharing of data or events between processors. Event sequences represent the order in which particular instructions, or operations, were executed on different processing nodes.

Communication between processors is vital to the analysis of program execution, since it represent pure overhead and results in under utilization of processing capacity. While communication cannot, in general, be eliminated it can be minimized through program design. Since processing nodes in many parallel systems operate asynchronously, global execution constraints must be enforced by explicit synchronization. The visualization of event sequences for different nodes allows this type of analysis.

Current research in parallel algorithm animation is focused in two critical areas: visualization/animation techniques for massively parallel machines, and development of an execution trace standard. The need for animations suitable for massive numbers of processing nodes is driven by the ever increasing dimension of parallel computing systems, required to solve computationally intensive applications. The display size of current workstations restricts the number of processors that can be depicted in any detail and thus more scalable depictions are sought. Trace standardization is important to allow portability of animation systems between computational platforms. This is vital and allows examination of programs when executing on different computer architectures.

The following sections examine the status of the major research efforts in the area of parallel algorithm visualization. In particular, currently available systems are discussed along with other relevant research papers.

2.2 Algorithm Animation Systems Composition

An algorithm animation system consists of a data collection/instrumentation component and a set of graphical displays that can be used to analyze this data. In addition to these fundamental components, the methods used to connect the instrumentation system and the graphical displays further defines the system. A typical implementation is shown in Figure 2. The analysis focuses on a user application program executing on a parallel computer system. As the application executes, event markers inserted into the program write trace data to the instrumentation system. The events can indicate information such as message transmission between nodes, function calls, program state, or other run-time program data. The instrumentation system gathers the events from different processing nodes in the system and can either send them directly to the display system, or write them to a storage file. The instrumentation system in many cases also provides global time ordering of events. The display system uses the ordered event trace to generate animation displays. The animation displays provide a visualization of one, or a number of program execution parameters. An example would be a utilization display indicating the number of active processors during different phases of the execution. Most systems allow a number of different display types to be active at one. The animation system usually allows for user input parameters such as time scale, the number of processing nodes, and trace file name.

The following sections discuss current issues relating to instrumentation systems and the visualization displays. Recent research has focused on improvements to data collection techniques, various graphical display formats, and the use of audio signals to complement the information provided by the visualizations (18).

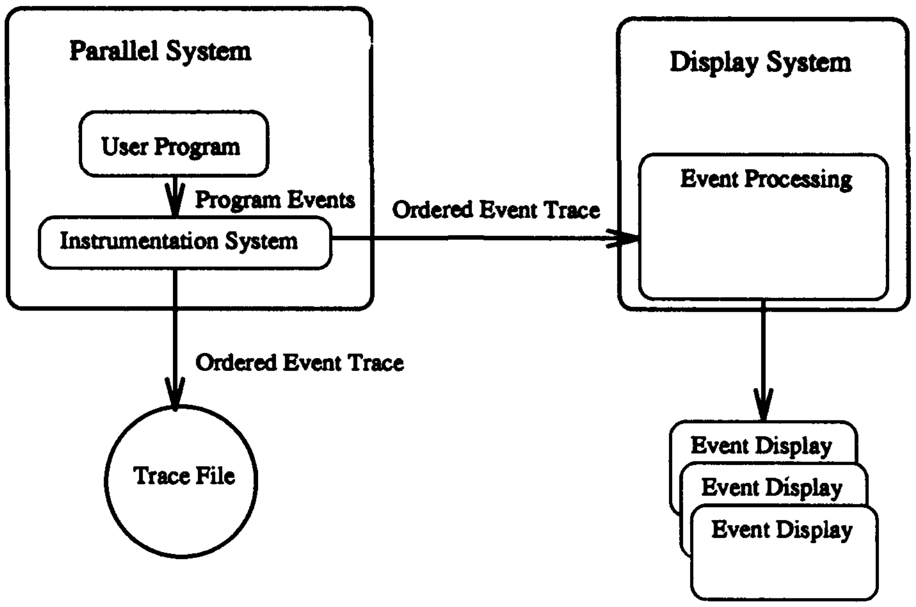


Figure 2. Typical Animation System Structure

2.3 Instrumentation Systems Issues

The execution of the program under examination is recorded as a series of event records. The records contain information relating to the event type, the time it occurred, and additional data structures of interest to the user. The recording of these events is performed by the instrumentation system. The instrumentation system must ensure that a relative global time is maintained between processing nodes, provide buffering of events, and produce file storage of trace data. In the AAARF system, data can be sent directly to the display system via Unix sockets. Since the data is distributed between processing nodes, the instrumentation system must be capable of combining separate traces into a single trace file. This is typically achieved by having an instrumentation control program on one particular node. This controller program then manages all trace file I/O activity. A typical instrumentation system is depicted in Figure 3.

2.3.1 PRASE: Data Collection System. PRASE (42) is a data collection system that is used to produce execution traces of parallel programs. This system was developed at AFIT and is specifically designed to collect data on the iPSC/2 parallel computer. The system incorporates the

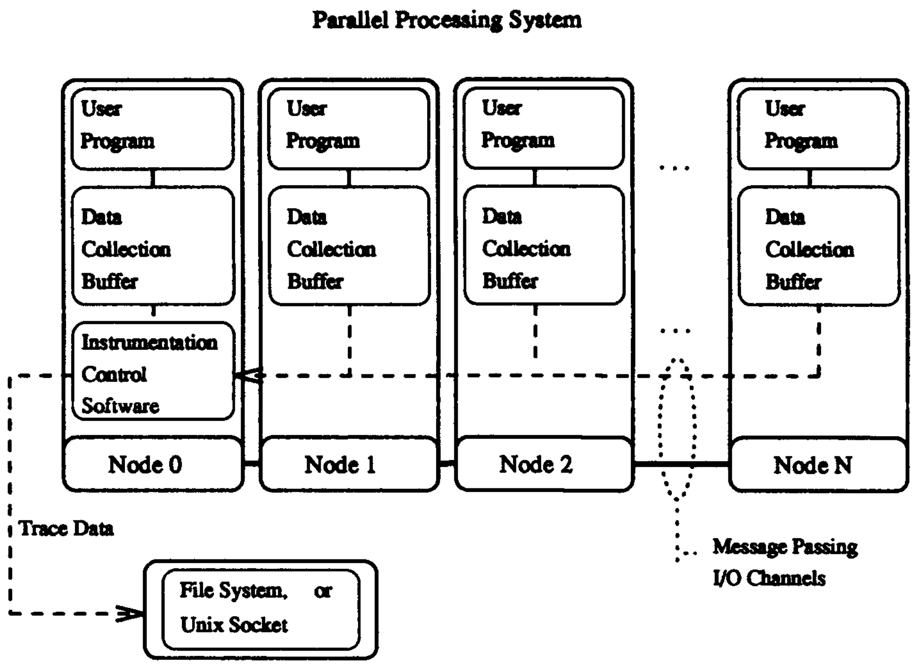


Figure 3. Typical Instrumentation System Structure

standard collection features that record inter-process communications and operating system calls as event records. These events provide the data required to generate low level process displays (84). In addition to these capabilities, the PRASE system enables the inclusion of more abstracted events into the execution trace. These events record higher level algorithm states (42) and program data structures. The trace data can be stored to a file or sent to a remote terminal over a communication network. PRASE is the instrumentation system used by the AFIT Algorithm Animation Research Facility (AAARF) (84). A overview of the PRASE data collection software is contained in Appendix A.

2.3.2 Reducing Instrumentation Data Volume. NASA's Ames Research Center has developed a system that reduces the volume of data collected by their instrumentation system (17). The reduction is achieved by the run-time analysis of execution events based on user specified requirements. The requirements are specified in the form of metric predicates. While this analysis places additional overheads on the program being executed, it can reduce the amount of trace data generated to less than one quarter of the original volume. The article also gives guidance on how to construct these predicates based on particular visualization requirements. The predicates are effectively user defined conditional statements that, when evaluated, determine if an event is to be recorded. For example, only message traffic to a particular node could be recorded when the communication delay exceeds some specified bound. This particular approach allows data collection to focus on specified abnormal behavior, assuming that the data required for analysis can be determined apriority. This type of intelligent event filtering provides one approach to the reduction in trace volume. Most current systems rely on selective tracing to reduce trace volume. Under this approach, only certain sections of the program, or sets of nodes, are instrumented. The metric predicate approach appears to be the only significant attempt at run-time trace data reduction.

2.3.3 Trace Data Standardization. Alva Counch's most recent paper on program visualization (12) outlines a proposed trace data standard. While this particular implementation has not gained wide acceptance, the concept is an important consideration for future visualization tools. The standard proposed in this paper outlines a trace data language that describes how events collected by the instrumentation system are to be translated into a normal form (12). The advantage of this normal form is that the use of a particular visualization tool is not constrained by the method used to collect the data. For the AAARF system to further develop, careful consideration must be given to conforming to a trace standard when, and if, one eventually emerges.

2.3.4 The Pablo Instrumentation Software. The instrumentation system developed for the Pablo (62) parallel program visualization tool provides a promising approach to trace format specification. This system uses a self defining event language that can be extended by the user to meet particular requirements. The system allows layering of these events to ensure control over data collection detail level. This is particularly important for controlling trace volume. Pablo's adoption by Intel for the Paragon (39) parallel processing system family, should ensure that it becomes a standard for these systems. The advantage of manufacture support and seamless integration into the OSF operating system provides a model for other system developers. The trace format utilized by Pablo is discussed in Appendix A.

2.3.5 Instrumentation Visualization System Connection. The majority of instrumentation systems collect trace data and store it as a text or binary file (43). The visualization tool then analyses the results off-line, after the execution of the program. This off-line analysis is prohibitive since it requires excessive amounts of data storage, even for short executions. As discussed in (84:4-9), the AAARF system overcomes much of this problem by implementing a real-time connection between the instrumentation system and the display workstation. The disadvantage with this approach however, is the resultant perturbations of the execution traces. For instances where detailed analysis of timing information is important, these perturbations make real-time data collection unsuitable. There remains a number of analysis problems where this approach is effective. Examples include programs with extended execution times, data structure visualization requirements, and program debugging. The ability to expand the run time displays to depict not only execution data, but high level algorithm information, is also highlighted (84:4-11).

2.4 An Overview of Visualization Systems

A visualization system translates the event records collected by the instrumentation system into animated graphical displays. The actual visualizations that are required by a user are dependent on both the type of algorithm under examination and the computational platform. All instrumentation systems for distributed systems are based on recording communication events. Depicting this information in a variety of animation formats is the central component of most animation systems. An example would be the utilization displays shown in Figure 4. This animation shows a histogram which displays the number of active and inactive processors relative to passing time. This display allows the user to determine periods of lost processing capacity due to message passing activity. The remaining AAARF animations are described in Section 3.4. These additional

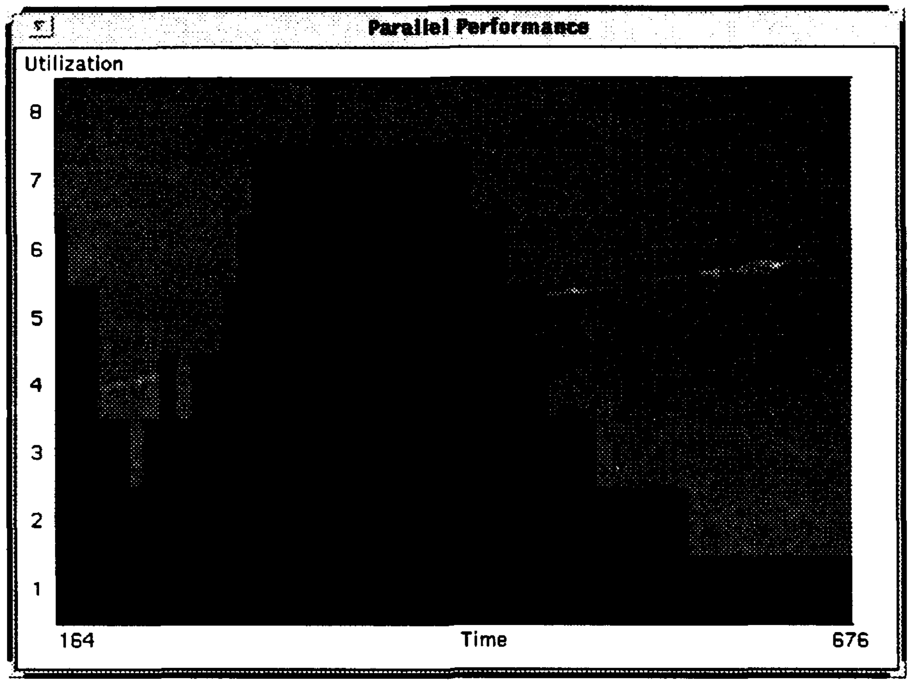


Figure 4. Typical System Level Animation Display

animations make it possible to determine other execution parameters that can be used to gain insight into program behavior. Based on this insight, improvements to the program's design or implementation may be possible. The following section outlines a number of current systems.

2.4.1 ParaGraph: A Visualization Tool. Paragraph is a visualization tool which displays trace data collected by PICL (23). The Paragraph system was developed by Oak Ridge National Laboratories and is discussed in (29). Paragraph is specifically aimed at distributed memory computer systems such as the iPSC Hypercube series. The Paragraph system has won wide acceptance in the academic community and is the most frequently utilized system in current research papers. The most important factor in this acceptance would appear to be its availability and simplicity rather than superior capabilities. Paragraph provides a range of displays that are aimed directly at the animation of process level statistics and the animation of low level program events. Paragraph includes only a very limited number of high level algorithm displays. These application specific displays are suitable for depicting small sorting problems, matrix transposition, and other matrix

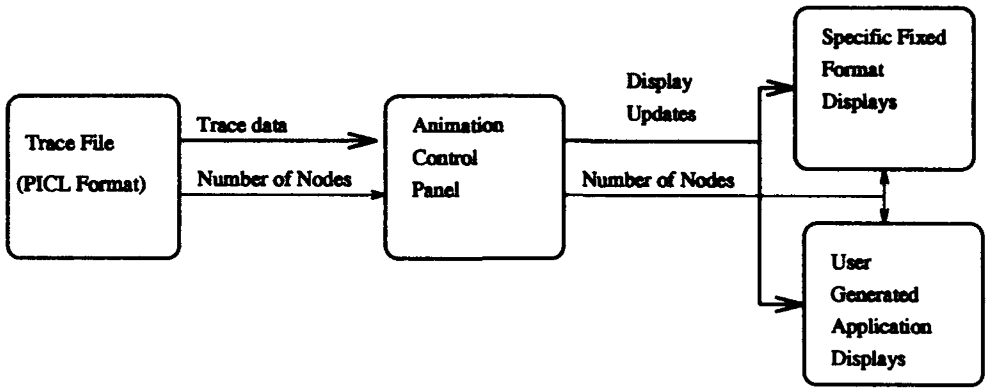


Figure 5. The Paragraph Animation System

computations. Paragraph has recently been incorporated into the Intel ParAide tool-kit for the Paragon computer system (39). This implementation has been modified however, to eliminate all user defined animations and operate using the Pablo (62) trace data format.

Figure 5 depicts the Paragraph system. The system only executes in the post-analysis mode, replaying previously recorded trace files. The system level events contained in the trace file map directly to aspects of the supported animations. The number of processing nodes is also recorded in the trace file and is used to scale the fixed format animation to the appropriate number of nodes. Users can add additional animations based on user specified or system level events. These additions require coding and re-compilation of the Paragraph system.

2.4.2 AIMS: The "VK" Visualization/Analysis Environment. One of the more extensive systems currently available for algorithm animation is the AIMS system which has been developed by the NASA Ames Research Center. The system consists of a number of component parts, firstly the Xinstrument system that produces instrumented code and the AIMS monitoring system that provides execution control and produces the event trace file. The trace file can be utilized directly using the AIMS VK visualizer, or using other utility programs, converted to formats suitable for Paragraph (29), Pablo (62), and Explorer (75). The most interesting aspect of the VK visualizer system is the level of association between the code, execution traces, animations and the instrumentation system. Events in animations can be interactively selected and the section of code associated with the event highlighted. The event types that are currently being displayed can be manipulated using an extensive range of event filters. The displays provided include histograms, gantt charts, and a space time diagram similar to Paragraph (28), however, more detail about the events is available on demand. For example, a particular message can be selected by the mouse on a time space diagram and the same event highlighted automatically on another animation. Under Paragraph the information is only depicted and user interrogation of a particular feature is not possible. One of the system's most powerful capabilities is the menu driven instrumentation system. The user is not required to directly edit the application program to insert the instrumentation code. The limitations of the system are its exclusive off-line operation and lack of user defined displays. However, this is compensated for by its ability to work with the other major visualization tools previously mentioned. AIMS is compatible with the iPSC/2 and iPSC/860 systems. A further limitation of the AIMS VK system is its inability to deal with extensive numbers of processor nodes. The current animations focus on systems with up to 300 nodes and are of limited use past this point. No attempt has been made to cluster separate processing nodes into combined display objects. Thus in most animations, every active processor is depicted. Under these conditions the display resolution limits the number of processing nodes that can be depicted. Figure 6 depicts the inter-operability features of the AIMS animation systems.

2.4.3 The Pablo Animation System. Pablo is an analysis environment designed to provide data capture and presentation across a wide variety of scalable parallel systems. The Pablo environment includes software performance instrumentation, graphical performance data reduction and analysis, and support for mapping performance data to both graphics and sound. Current extensions to the Pablo system include data immersion via head-mounted displays and integration of High Performance Fortran (HPF) compatibility. Pablo's key features are the previously described self-defining event formats and user configurable display formats. Rather than providing

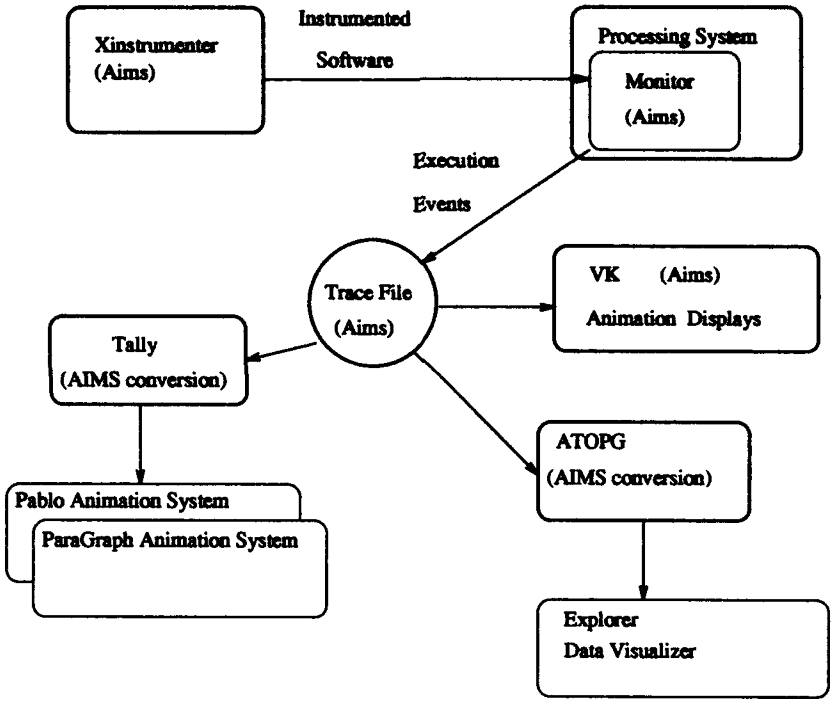


Figure 6. Block diagram of the AIMS Parallel Animation System

only fixed format displays, the concept of user configurability significantly improves Pablo's ability to deal with application dependent animations. The graphical performance analysis component of the Pablo system consists of a set of data transformation modules that can be graphically interconnected. These modules provide user specified data transforms on the incoming trace data and map the resultant output to other modules of display animations. Pablo data analysis animation displays include bar graphs, bubble charts, strip-charts, contour plots, dials, interval plots, kaviat diagrams, scatter plots, matrix displays, pie charts, and 3-dimensional scatter plots (55). A block diagram of the Pablo system is shown in Figure 7. The graphically composible transform modules can be connected to form an acyclic graph. This graph structure defines how event formats are mapped to performance parameters. This mapping can also include operations such as sorting, counting, minima, maxima, powers, trigonometric functions, and simple statistics. The resultant performance parameters are then mapped to specific display formats. The important aspect of

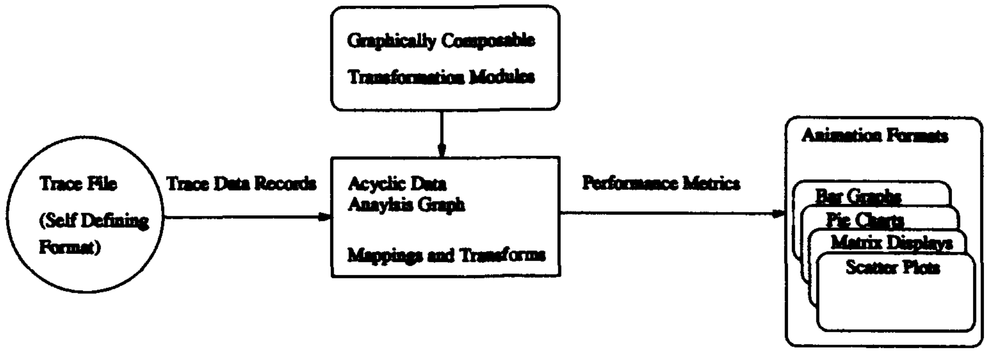


Figure 7. The Pablo Animation System

these display formats is that they are independent of the number of processing nodes and parallel system architecture.

The system was designed to specifically tackle the problems of portability, scalability, and extensibility. It achieves these objectives very successfully and is currently the most suitable for work with massively parallel systems. Pablo is best viewed as a toolkit for the construction of performance analysis environments. The system is a continuing work and under constant change. The only major weakness from an algorithm animation perspective is its poor support for generation of arbitrary display formats. Under the Pablo system, users utilize generic display formats and produce mappings for interesting parameters. In many of the algorithms examined in this research, animations generated from visualization of distributed data structures require more flexible display formats.

2.4.4 ParAide: Paragon System Development Toolkit. One of the best indications of the success of parallel algorithm animation systems is their adoption of these tools into the latest version of Intel's parallel program tools set. The ParAide tool set was released in the last quarter of 1993 and includes a range of development, animation, debugging, and system utilization tools for the Paragon system. The system does not currently support the iPSC/860 systems. Intel has included in ParAide a modified version of the Paragraph system (28) and has converted it to be compatible with the Pablo (62) trace format. This modification has resulted in the elimination of application specific animations which were included in the original version of Paragraph. While the Paragraph component of the tool remains an off-line animation, the displays are complemented by a real time display provided by the system performance visualizer (SPV). SPV provides a utilization and node allocation display for the Paragon system. The performance statistics are updated every second via Unix socket communications with the user display terminal. Both processor utilization and message passing activity can be observed using this tool, including processor bus activity. Its primary purpose is to allow users to determine system status and is only suitable for analysis of program execution in a gross sense, due to the low time resolution. Of greater importance is the acceptance by Intel of a particular trace format. This should at least provide commonality for researchers utilizing Intel products. Intel plans to extend the ParAide animation suite to include the Pablo (62) system. Pablo is possibly the most advanced of the current animation systems and its inclusion will provide greater capacity to deal with massive numbers of processing elements. The integration of the interactive debugger and the animation system, present in ParAide, is certainly an important development feature. Actual utilization of the ParAide tool was possible and the ability to control the execution of the target program during animation was of significant benefit. If visualization tools are to be effective for program debugging, they must allow execution control of the program under examination. The combination of program visualization and code level debugger with execution control make ParAide an effective development tool. However, the lack of application specific displays and data structure animations limits its use in algorithm development. In fact for this research, ParAide provides no tangible benefit over the AAARF system.

2.4.5 VISTA - Visualization and Instrumentation of Scalable mulTicomputer Applications. VISTA is an instrumentation and visualization paradigm (67:1) specifically tailored to directly address scalability. The VISTA paradigm treats performance data essentially the same as the distributed data of the programming models used for parallel programming. VISTA focuses on views that represent the physical or logical layout of the processing nodes. Color coding, or additional display objects are used to depict actual performance statistics. The statistics that can be visualized by VISTA include execution time, operation count, computation time, message volume, communication time, communication flow, communication overhead, and other percentage summaries of these factors.

The VISTA paradigm is composed of three components: Visualization, Data Parallel Representation, and Performance Measurement. The basis for the Visualization component is the state of a Processing Element (PE), which translates to one or more quantitative metrics. For example, the performance parameter could represent the total message volume for a particular processing node. The visualization component is divided into four levels:

- *microscopic snapshot.* A performance parameter K at some specific time on a particular processor.
- *microscopic profile.* A microscopic snapshot which allows K to vary over time. (An AAARF or ParaGraph style single processor display).
- *macroscopic snapshot.* A microscopic snapshot of all PEs at a specific time for a particular K. This forms a two dimensional mapping and is essentially the same idea used by AAARF and ParaGraph for multiprocessor displays.
- *macroscopic profile.* A macroscopic snapshot allowing K to vary over time.

The four levels are referred to as Machine Views, and are instances of a general class of multivariate data plots tailored to display performance measurement data (67:5). The snapshot views are simply the instantaneous value animations (IVA), used in AAARF and many other systems (43:2-5). They provide a static display of a set of performance statistics, the display being totally updated with new information at regular intervals. The profile view corresponds to the windowed time interval (WTI) displays, also used in AAARF, and other systems (43:2-5). These animations are usually presented in a scrolling format, with sections of the display reflecting parameters at different points in time. They effectively provide a short history (the length of which is dependent on the time interval displayed in the window) of particular performance parameters.

2.4.6 MTOOL: A Shared Memory Visualization Tool. Stanford University has developed a system for visualization of shared memory systems (24). MTOOL is compatible with only two processing systems, the Stanford DASH (24:490) (an in house experimental system), and the Silicon Graphics multiprocessor systems. Reported experimentation with MTOOL relates to monitoring eight processor Silicon Graphics Inc. (SGI) 380 shared memory multiprocessor systems. MTOOL is primarily a visualization tool, but includes the instrumentation system as an integral component. The instrumentation system is operated via the display system. This system provides automatic insertion of the instrumentation code into the user's code and adds only a 5% execution overhead (24:481) to the program under investigation. In contrast to other systems, MTOOL does not animate the execution of the program under test, but provides statistical summaries in the form of graphs for particular sections of program execution. The system is under constant expansion to include more insight into the problems of poor memory performance, a major consideration for shared memory systems. The most interesting feature of the MTOOL system is its focus on examining synchronization overhead, memory hierarchy performance losses, and the extra work required in parallel programs (vs. sequential). The majority of other systems including AAARF and Paragraph, consider the node as a simple processing element and internal performance of the processing nodes is not considered. Optimization activity often focuses on memory allocation and compiler optimization and thus this area is increasingly important to overall parallel performance.

2.4.7 Visualization Enhancement with Sound. Researcher's at Michigan State University have recently examined the use of sound to improve the user's comprehension of the visualization displays (18). The article examines the association of audio signals to a number of performance parameters such as processor utilization. The most promising application is to alert the user to some important global events when visualizing detailed displays of low level data. While this approach can only augment the visualization, as highlighted by the authors (18:435), it provides an additional information channel to the user. Its use in focusing the user's attention to particular aspects of the animation is also discussed. Limited use of this enhancement is appearing in other research (33).

2.5 Summary

The goal of visualization research is to provide abstract views of the program's execution, which require manageable instrumentation data requirements. The most widely utilized animation systems, Paragraph, Pablo, AIMS, and ParAide all approach this goal in different ways. Paragraph and ParAide are effectively the same and represent the older generation of systems. The fixed format displays provided by these systems are informative and flexible for analysis but are inherently un-scalable. They rely on strongly formatted events that can be mapped directly to display formats. AIMs represents an improvement over Paragraph in that it allows interoperability with data visualization tools. The visualizations contained in the VK animation component of AIMs differ only in format and features from the Paragraph depictions. AIMs does, however, allow the direct association of program instructions to animation displays. Pablo more directly tackles the problems of scalability and portability. Pablo's system independent animations and trace format provide scalability to meet the demands of massively parallel processing systems. The self defining trace format also provides a basis for portability of the Pablo system to other parallel architectures.

The ability of programs to utilize the processing capacity of parallel computer systems may well depend on the development of effective visualization tools. Visualization tools provide insight into the complex execution patterns of parallel programs. This insight can be used by the program developer to enhance or correct his design/implementation. In addition to providing insight into the execution of parallel programs, animation systems can present quantitative data detailing execution parameters such as concurrency level, utilization of particular processing nodes, communication volume/load, and synchronization overhead. While a number of systems for visualization currently exist, the majority are utilized exclusively by the developing organization. The ParAide system (39) provides the most promising indications that these animation systems will become more widely accepted by application programmers.

While much has been achieved, commercial visualization systems such as ParAide still fail to meet all animation objectives. For example, many of the views provided by ParAide are incapable, due to scalability problems, of depicting the execution of programs on even medium sized Paragon systems. In addition, the animations remain focused on depicting low level system performance information, at the expense of abstract algorithm data. These low level animations depend on detailed event traces that are unlikely to be possible on massively parallel system.

The approach taken with AAARF in this research departs from these low level animations to focus on more abstract depictions of program behavior. This focus enhances scalability, while reducing trace data requirements. It also allows realistic application of the real-time animation capabilities of the AAARF system. To increase the use of animation tools within AFIT, this research expands the capabilities of the AAARF system. The next chapter describes the initial improvements made to the generic system level animations provided by AAARF.

III. Animation Design, Synthesis, and Analysis

3.1 Introduction

The displays provided by current animation systems are largely application independent as discussed in Chapter II. This ensures that new applications can be instrumented with the minimum of effort. Animations of this type are usually generated from inter-node communication traces and do not require knowledge of algorithm data structures. However, this limits the animation to focusing on the algorithm's control flow, as it is reflected in this communication. This section examines the development of additional application independent displays for AAARF that deal with this problem. Animations included in other systems are evaluated and their effectiveness is examined using a variety of linear algebra algorithms. To expand the educational usage of AAARF, additional algorithms have been instrumented and a library established.

3.2 Instrumented Algorithms

AAARF was originally developed to provide an animation environment for serial programs by Fife (16). The potential and flexibility of the original design resulted in its selection by Williams, for use in parallel algorithm animation. The AAARF system was first utilized for parallel algorithm analysis by Williams (84) who instrumented a number of parallel algorithms during development of his extensions to the AAARF system. The Car-wash simulation (83) was added later by Wright (86). An instrumented library consisting of the following parallel programs had been established at commencement of this research activity.

- Set Covering Program (SCP).
- A Parallel Bitonic Merge.
- The Car-wash Spectrum (64) based Parallel Discrete Event Simulation (PDES).
- Intel Ring demonstration program (72).

While this proved the utility of the AAARF system, the work by later researchers Lack (44) and Wright (86) has focused on establishing a more stable implementation. The X windows version of AAARF developed from the original SunView, by Wright, now provides a system suitable for further algorithm research. To further expand the range of algorithm demonstrations available for educational purposes, a number of programs have been instrumented.

- Fast Fourier Transform. (45).
- Matrix multiplication. (45).
- Guassian Matrix inversion (45).
- LU Matrix decomposition (45).
- Three dimensional Wavelet Transform (45).
- Mission Routing A* search (14).
- Genetic Algorithm (Molecular Bonding) (6).
- VHDL parallel discrete event simulation (5).

These programs provide a varied range of applications that cover many classes of algorithms. The fast Fourier transform, matrix multiplication, guassian inversion, wavelet transform, and LU decomposition are all data decomposition implementations of mathematical transforms. The mission routing implementation is a distributed search algorithm using the worker/controller paradigm. The genetic algorithm application is also a data decomposition algorithm, however, it has a non deterministic execution behavior as a result of its probablistic search technique. The parallel discrete event simulation is representative of algorithms with non deterministic control structures. This diverse range of applications allows examination of AAARF displays on real programs. These programs are now stored in the /usr2/aaarfDEMOS/ directory and can be used as examples by future parallel research students. Instruction on their compilation and execution requirements are contained in the AAARF users guide (58).

3.3 Generic Animation Displays

This section examines the utility of the current generic animation displays when applied to analysis of the programs previously discussed, the overall objective being to improve the current animation ability of AAARF from both software development and educational perspectives. This investigation sought to determine improvements to the AAARF system that provide meaningful insights into the operation of other AFIT parallel research projects.

3.4 Analysis of Important Animation Criteria

An algorithm can be considered to be a combination of a set of data structures D and a set of events (or operations) E. The definition of the algorithm would also include a control sequence S for the event set E. From an animation perspective S is unimportant, since the animation is showing the execution of the program, not controlling it. Thus for animation, we must focus on the data structures and the events. This concept for animation was discussed by Fife (16) and was elaborated by William (84). In a parallel implementation these events and data are distributed across a number of processors. This distribution results in additional events and effects being introduced as a result of the program's mapping to the parallel system. The actual mapping of a particular algorithm to a parallel architecture is an optimization problem that is the focus of much research activity (47, 70, 10).

We can build on this basic definition by considering the combination of this information into the higher level more abstract information. This allows us to consider the program from a number of perspectives:

- System level events.
- System level performance.
- Algorithm performance.

- Algorithm Data Structures.

- Algorithm Events.

System events and performance data allow us to determine how well we are utilizing the capabilities of the target hardware. Events simply reflect the occurrence of specific actions. An example would be the completion of a message transmission. Performance information relates the occurrence of events to the passing of time. An example of this type of information would be percentage utilization figures generated from analysis of active/idle events. System level events are the focus of the majority of parallel animation systems including AIMS, Paragraph, ParAide, VISTA, and Pablo, discussed in Chapter II. In particular this information is gathered by recording communication events in distributed processing systems. By recording only these events a significant number of basic trace data can be produced. These include:

- Periods of processor activity/inactivity.

- Inter-processor execution dependencies.

- Inter-processor message traffic.

Based on this generic data it is possible to determine a number of important execution parameters. In particular, we are able to examine utilization of both the processing capacity and message passing bandwidth. These two characters can be depicted in a virtually unlimited number of animation views. The data can be presented directly, averaged, summarized, or transformed into an abstract representation. AAARF provides ten system level views which are common to most animation systems. Animation research by other researchers has produced a preponderance of other animation formats for system level events. While the number of views possible are extensive, the parameters of interest at this level are not. From the data collected using the generic instrumentation of message passing events, we aim to extract the following information:

- Processor utilization both during particular phases and as overall statistics.

- Communication loading both during particular phases and as overall statistics.
- Time synchronization of events on different nodes.
- Processor interplay and dependencies.

From depictions of this information we aim to better understand the behavior of the algorithm when executing on a particular parallel computer system. With this information, modification and enhancement of both algorithm design and implementation can be achieved. In most cases our overall goal is to reduce the execution time required by the algorithm to produce our computational results. This in turn is achieved by gaining maximum *effective* utilization of processing capacity. It is important to note that these displays are all generated from tracing message passing activity. No information relating to the actual on node control flow, or data structures is provided. To provide this information, AAARF originally contained the following animations:

1. **Animation:** Processors are depicted in a circle. The color of the processor indicates its status- idle, busy, blocked, or sending. Lines are drawn between processors to indicate message activity.

2. **Kaviat:** The processors are located at the perimeter of a circle and spokes are drawn from each processor to the center of the circle. The CPU utilization for each processor is calculated and plotted along the corresponding spoke of the wheel, with the center representing 0 %.

3. **Utilization:** This view is a histogram which displays the number of active and inactive processors on the vertical axis, and time on the horizontal axis. The active vs idle state of the processor is derived from the message traffic: if a processor is blocked waiting for a message, then it is idle. Otherwise the processor is considered active.

4. **Feynman:** This view shows two different types of data; processor status and message passing activity. The processor status is shown in a similar fashion to the Gantt view, except that the data is shown as a horizontal line that is broken when the processor is blocked. Once

again, the horizontal axis represents time. The message activity is shown by drawing lines between the horizontal lines representing the sending and receiving processors. The ends of the lines are positioned according to the send and receive times. This clearly shows the message passing patterns, delays, and bottlenecks. It is particularly useful in determining critical paths in algorithm executions.

5. **Communication Statistics:** This view provides a detailed look at the communications activity of a single (selectable) node. The data displayed can be chosen from three types: processor source or destination for each message, length of the messages, or the message type.

6. **Communications Load:** This view shows statistics on pending messages for the entire system versus time. Either the number of pending messages or the total length of all pending messages can be displayed.

7. **Queue Size:** Statistics for the input queue for one (selectable) node are displayed in this view.

8. **Message Lengths:** This view uses a different method of presenting message passing information. The view contains a matrix, and a send operation causes an element of the matrix to be colored. The sending processor determines the row and the receiving processor determines the column. The length of the message determines the color.

9. **Message Queues:** This view displays the current status of the input message queues for all the processors under examination. The display is a histogram with the processors along the horizontal axis. The vertical axis can show either the number of messages in the queue or the total length of the messages in the queue. As the queue levels rise and fall, a "high tide mark" is left behind to mark the highest level that was attained for the run.

Examples of these displays are contained in the AAARF users guide (58). All the above mentioned displays are generated from event records that include only the following information:

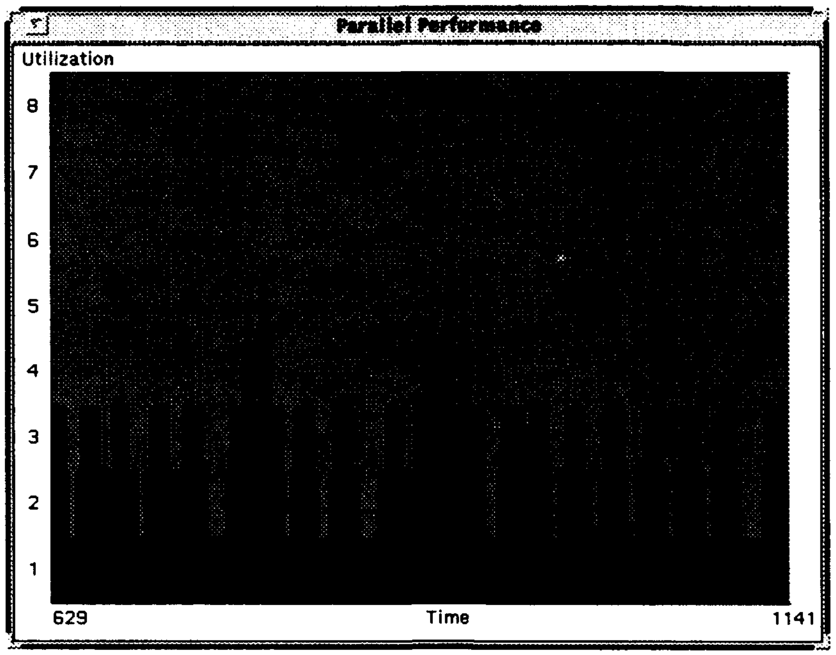


Figure 8. Utilization Animation of a Matrix Multiplication Algorithm

- Time send command executed.
- Time receive command executed.
- Timed message received.
- Message type identification.
- Message size in bytes.
- Source processing node.
- Destination processing node/s.

By instrumenting programs with the basic PRASE event tracing we can readily animate many aspects of the program's performance. This should be the first step in developing any animation since it provides preliminary information that can guide development of application specific displays. The low level event tracing of processor utilization and inter- node communication are easily handled by the standard animations provided in AAARF. Firstly, the utilization display (Figure 8) clearly highlights the implementation's ability to utilize the allocated processors. In this particular example

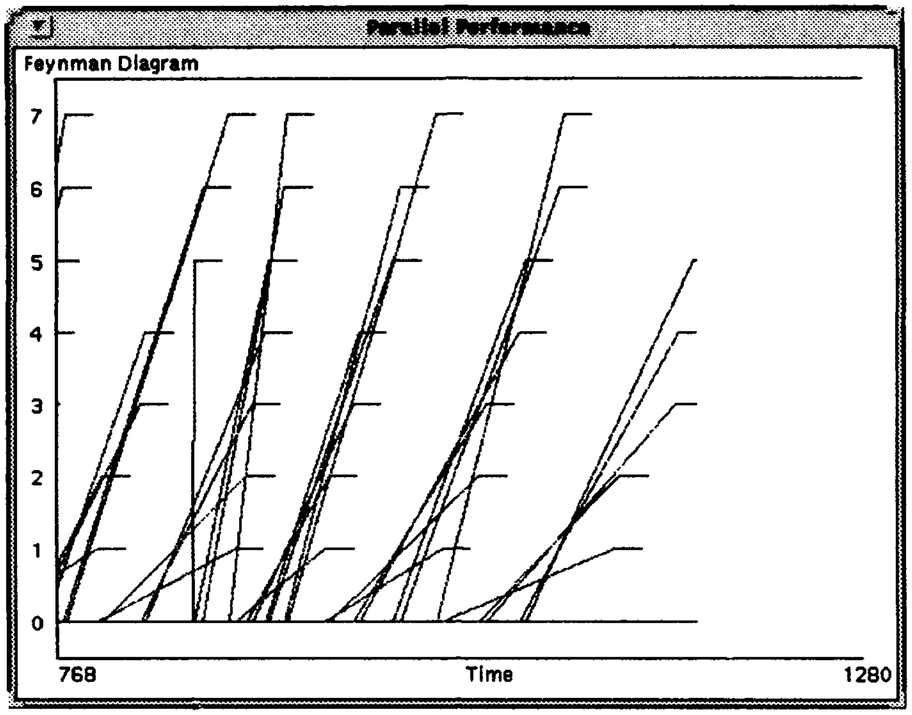


Figure 9. Feynman Animation of a Matrix Multiplication Algorithm

only two to three processors are on average being utilized. The Feynman diagram (Figure 9) shows the effectiveness of the control structure decomposition and any load imbalance. In this particular display it is possible to determine message dependencies and processor idle time due to these dependencies. However, in more complex algorithms, without complementing these displays with specific data structure animations, the reason for these execution patterns is impossible to determine.

3.4.1 Deficiencies in current AAARF Animations. The AAARF system proved to be an invaluable tool in analyzing the execution patterns of the algorithms listed in Section 3.2. In particular, the information presented by the Freeman diagram (84) provided instantaneous recognition of algorithm behavior. It was particularly useful when examining existing programs that were not developed by the user. In this code reuse analysis type application, the display revealed a considerable amount of information about the program's structure. This includes information about event synchronization, and process interdependencies, how data was shared between processing nodes, and the execution time cost of this activity. While useful for nearly all applications, the current displays had a number of limitations, in particular, the lack of overall statistics displays for both utilization and communication events. The following sections outline a number of improvements appropriate to the current generic displays.

3.5 Message Color Coding

When examining the communication patterns of an executing program, the message type is displayed on the status panel. This status panel only reflects the type of message last received. Thus in the Feynman diagram for example, all inter-node communications appear as black lines. This makes interpretation of the message traffic difficult and the control structure becomes hidden in the shear volume of data. To regain this information, a method for distinguishing different message types is required. This can be achieved by modifying some aspect, or aspects, of the lines used to represent the communication events. While labeling, line type, and thickness could be varied, the use of color coding gave a clearer representation.

This particular feature is included in other systems such as PICL. In the case of many programs, the message type indicates the commencement of different phases of the program execution and thus indicates causal relationships. The AIMS (57) further extends this concept allowing user interaction with the trace and subsequent depiction of the relevant code segment. While this initially appears appealing, it provides only a marginal improvement of the information contained in the AAARF status panel. Relating a particular message to a code segment is, in most cases, relatively simple since the message types are uniquely identified. This level of activity is more relevant for functionality included in program debuggers where execution control is possible.

3.6 *Phase/Task versus Time Animation*

A well-designed program consists of a number of separate functions. Whether the design is object oriented or functionally based, certain procedures and functions relate to identifiable program requirements. These requirements may reflect either required problem domain behavior, or implementation dependent operations.

Under the functional programming approach, the functions represent transforms that are applied to the program data. The order in which these functions are applied is specified in some form of task graph (70), and the program's control structure enforces this sequence. These functions can be considered as tasks or jobs that must be performed in order to obtain the desired program behavior. Thus, depiction of functions calls can provide insight into their order of execution, frequency, duration, and distribution across the processing nodes. This can be particularly useful when examining the results of task scheduling, or allocation activity. These are significant research areas that have a large impact on the ability of users to design optimal parallel program implementations. Both aim to execute the functions contained in a parallel program, in such a manner as to minimize execution time.

With an object based implementation, functions and procedures are utilized as object methods (69). In this programming paradigm, monitoring function calls allows depiction of methods that are evoked on different objects during execution. Since in this paradigm, execution control results from events generated by object updates, it is important that the user can observe this behavior. Thus, independent of the program design paradigm, function calls relate to specific actions that are required to complete the assigned computational task. Since these functions are often designed to reflect real world operations, they represent a user interpretable program activity. Both the functional and object based approaches provide a level of implementation abstraction by encapsulating operations into functions. This can be readily utilized by the animation system.

The Pablo, AIMS and Paragraph animation systems provide task based visualizations. Pablo in particular tends to emphasize program profiling aspects of this animation. All three systems use Gantt type displays (84) to represent the individual processors executing different program functions. This basic approach has been incorporated into AAARF.

3.6.1 Implementation in AAARF. The AAARF implementation of phase/task displays allows the user to observe in a Gantt chart format, the commencement and completion of program function calls. The AAARF implementation allows greater user flexibility in that the color coding of functions is not fixed. In the Paragraph (28) system, each function is allocated an integer value and a color code. Each function is then represented on the display. The AAARF implementation can provide this type of representation but also allows different functions to be treated as a single entity. In this way the AAARF implementation allows the user to group functions based on the area of interest. For example, all functions related to linked list management could be highlighted against all other program activity. In this way, the overall effect of this program characteristic can be examined. If further investigation is warranted, the different linked list functions could be assigned different colors relative to all other program functions and examined in more detail.

This flexibility allows the user to not strictly consider the program as a combination of basic functions. but as a collection of higher level activities or phases. A example of the phase animation is shown in Figure 10. This particular example on eight processing nodes, is taken from the Mission Routing research by Droddy (14)

3.6.2 Process timing. The analysis of the programs listed in Section 3.2 highlight the need to time certain events. For example, the time required to execute a particular function may depend largely on the data passed. In a situation where this activity is central to the algorithm's execution, the average execution time and its variance provides the user with important information. This data allows determination of scalability and the ability to predict performance on other systems. In

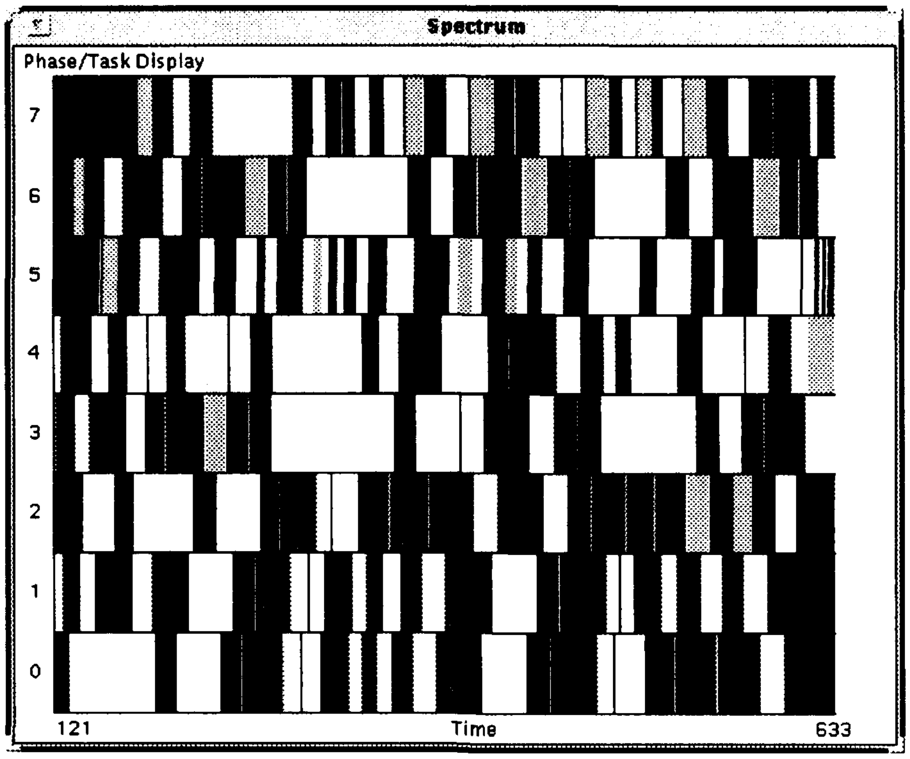


Figure 10. Phase Display - Depicting program execution characteristics

this case, the use of a scrolling bar graph display provides a timed record of previous occurrences, along with the maximum, minimum, and average values.

The animation allows the user to insert interesting event (IEs) marks at the beginning and end of code segments under examination. The user is not constrained to measuring a single characteristic but can insert any number of marks into the target program. During algorithm analysis, the user can select which are to be displayed. Figure 11 provides an example of this display showing the time required to perform population fitness evaluations in a genetic algorithm program, discussed in detail in Chapter X.

3.7 Further System Level Displays

The additional animations discussed in this chapter enhanced the capabilities of the AAARF system. Based on the original objective of providing additional support to other AFIT research

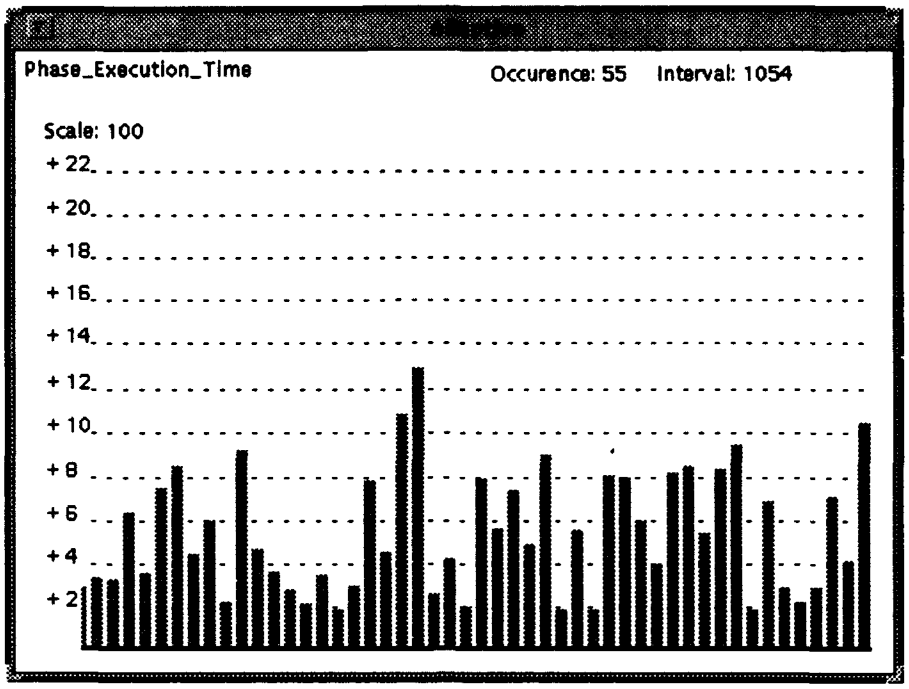


Figure 11. Process Timing - For Genetic Algorithm population Fitness Evaluations.

projects, the new animations are successful. In particular, the task/phase displays provide insight into the execution of the program on the individual nodes. Previously, actual execution steps on the nodes were not depicted and this addition reveals this aspect to the user.

The three additional animations outlined in this chapter were initially considered to represent a fruitful path for further development activity. This conclusion was based on the ability to generate a considerable number of alternative views of the same basic trace data, each particular variation casting a different perspective on the underlying trace data. Trial use of other systems including AIMS (57), Pablo (62), and Paragraph (28) revealed further displays that could be included in the AAARF system. Paragraph alone includes an additional 20 display formats not currently contained in AAARF. This does not include possible layout variations for each of these additional animations. For example, the Paragraph equivalent of the AAARF animation display (Section 8.6) includes 10 variations to account for processor layout format (ring, mesh, grid, hypercube, etc) alone. However, this initially fruitful area of experimentation fails to satisfy our objectives. For reasons outlined in the following chapters, the concept of fixed format displays is too restrictive to meet our animation objectives. The addition of further fixed format displays provide only marginally more information than the current AAARF animation and still leave a significant portion of the execution hidden from the user. As highlighted in Chapters IV, V & X, application specific displays that present algorithm data and control structures provide a significant leap in capability. The demands of these application specific displays require animation development environments, not fixed format implementations. To this end, Chapter IX discusses the development of an adaptive animation environment that allows the construction of arbitrary animations without code generation. By constructing a display definition file, the user is capable of generating virtually all animations contained in other systems without coding. For this reason, further research activity was focused on this work rather than minor variation to the current set of AAARF animations.

Another significant result of this experimentation was the realization that the current AAARF system level displays are too narrowly focused on run-time analysis. These run-time displays depict instantaneous, or short histories for the observed performance statistics. However, this does not satisfy the requirement for displays that can summarize overall performance, or highlight periods of execution that warrant more detailed examination. A good example of this is determining the message distribution across system communication channels during program execution. In this case, the actual run-time temporal behavior is relatively unimportant and only a completion summary report is required. Relative message traffic volumes on physical interconnections are more important for determining the suitability of a particular program decomposition. This observation gave rise to the development of integrated analysis displays which are discussed in Section VI. This tool aims to provide static displays that assist program development activity directly.

3.8 Summary

The research discussed in this, as well as later chapters, highlights the futility of producing additional generic animations. Only the lack of execution summary displays presents a shortcoming in the present AAARF implementation at the system level. The development of this capability is discussed in Chapter VI. Beyond this addition, the magnitude of improvement and the level of insight required for analysis of program execution demands the use of application specific displays. The following chapters apply application specific displays to a number of AFIT algorithm research projects. The next chapter in particular, addresses animations for mission routing research (14). This work culminates in the development of an adaptive animation generation environment.

IV. Animation of Parallel Mission Routing Algorithm

4.1 Introduction

AFIT has been extensively involved in search algorithm research (1, 14, 35). Current research includes the development of an A* search algorithm (59:63) for aircraft mission routing (35). This section examines the integration into AAARF of a user interface and application specific displays for this type of algorithm. The application specific displays and event definitions developed for this application can be readily used for other search algorithms. This section clearly shows how application specific animations can be used to enhance algorithm development. In particular, animations that allow determination of actual algorithm performance, where execution time is inappropriate, are developed.

4.2 Introduction to AFIT Multi-criteria Aircraft Routing Problem

The mission routing problem consists of the selection of a route to a target that will ensure the greatest probability of success at the lowest possible cost (risk, energy, etc). The probability of mission success is calculated by determining the radar exposure produced by the route. The m cost is simply the total distance traveled. Further, the problem can be generalized to incl additional requirements reflecting multi-criteria optimization. This problem has been examined with parallel implementations by Grimm and Droddy (35, 14). The current implementation searches a three dimensional grid within which both terrain and radar sites are modelled. The desired mission destinations, radar sites, and terrain maps are contained in files that must be passed to the A* search program.

4.3 Performance Statistics Displays

The total execution time required to produce a solution is often used as a measure when evaluating different algorithms or various implementations. While the execution time provides a reasonable figure-of-merit for many problem classes (e.g linear algebra), the execution time for search algorithms tends to be very problem dependent (4). Even with identical problems, a small change in the search algorithm may result in large changes in performance which may not reflect the general case (59). Subsequently, it is important to consider a number of higher level primary measures to gain insight into the reasons for a particular performance result, the aim of new displays being to highlight features that allow the user to determine algorithm performance on similar problems and other parallel system architectures. The total number of nodes expanded during the search process differs from one execution to another execution, even with the identical problem and system configuration. This results from the non- determinism introduced by message passing between processor nodes, since the processors are not synchronized. Thus, for load balancing and node expansion optimization, total execution time does not provide an accurate measure.

To more accurately measure performance under these algorithm conditions, execution time must be related to nodes expanded, not just solution time. The following parameters were considered to be of interest.

- *Search space nodes expanded per unit of execution time.* Depiction of this information allows determination of aggregate performance measured in search space nodes examined. (Nodes per Unit Time).

- *Rate of node expansion per system processor.* Depiction of this information shows the effect of load balancing between processing nodes and reveals the distribution of effort for the measured aggregate performance. (Expansion Rate at Processor).

- *Computational time per program phase (Phase Time).* As a result of its repetitive execution pattern, small improvements in frequently executed functions can provide significant increases in performance. The execution costs of different program phases can be determined with visualization of this information.

The following displays have been implemented to meet this requirement. These displays complement processor utilization figures produced by standard animations, but have the advantage of relating more directly to the user's problem space.

4.3.1 Nodes per Unit Time Animation. This display allows the user to determine the average time required for a node to be expanded by the system. The animation depicts, using a scrolling bar display, the total number of search space nodes expanded by the system. This animation provides a graphical indication of the current average time required to expand a search node on a system processor. The information is presented as instantaneous value animation in a bar graph format. The result of grain size experimentation can be evaluated using this display by determining changes in the node expansion rate. This is important when determining the unit of work to be allocated to the processors for optimum performance.

4.3.2 Expansion Rate Animation. The expansion rate display shows how much effective progress is being made by individual processor nodes by displaying a bar graph representation of nodes expanded by each processor node. This display is particularly useful in determining if the algorithm is evenly balancing the work load amongst the available processors. This allows the evaluation of load balancing strategies (14). In particular, the bottle necks introduced by the controller/worker paradigm are made clearly apparent. With relatively small numbers of nodes the *controller* is able to effectively utilize all *worker* nodes. As the number of nodes increased, (> 16), many workers became under utilized. This particular animation allowed different work allocation strategies to be examined visually.

4.3.3 Phase Time Animation. The search process can be considered to consist of different phases that are repeated as nodes are expanded in the search space. The phase time animation provides a Gantt chart animation of the interesting phases of the program's execution. Each node processor continually executes the following basic loop in the mission routing program (14:5-12).

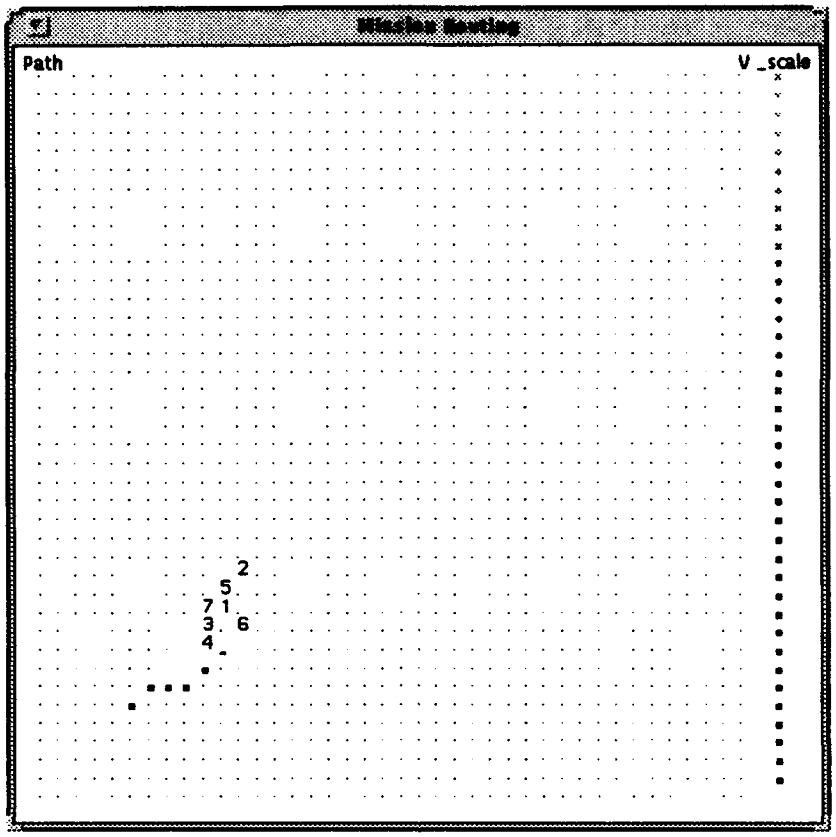


Figure 12. Current Best Path Display with Processor Locations

```
While Node to expand
    Get next Node
    Generate Children
    Calculate new cost(Radar, Displacement)
    Insert New Nodes in Linked List
    Send selected Nodes to Controller
End
```

By color coding these phases, the computational cost of list insertion, radar exposure calculation and other functional components can be compared. This is a particularly useful display as compared to the simple function call animations provided by other systems. In this case, the combination of many functions into logical groupings helps to abstract unnecessary detail. This approach is useful in providing a basic profiling function for parallel programs. This can highlight sections of code that would benefit from better coding and give a clearer indication of the impact of different data structures on algorithm performance.

4.4 Animation of Search Space

The most common animation for search algorithms is a representation of the underlying control and data structures. In the case of the mission routing problem, the use of search space representations provide useful insight. In the case of the A* search algorithm, displaying of data structures does not reveal the execution of the program, since in this case, the central data structure simply defines the search space. However, the partial solutions and the nodes selected for expansion, reflect the current status of the algorithm.

4.4.1 Search Space Representation. The search space examined by the mission routing algorithm is a three dimensional grid which contains terrain and radar features. Three animations of the search space were generated. Figure 12, 13 & 14 depict the current best solution, nodes expanded, and the current areas being searched by the individual processors.

4.4.1.1 Current Best Path. The first display is depicted in Figure 12 and shows the current best path of the nodes being expanded by the processors. The display uses color coding to depict the vertical dimension. The status panel shows the cost of the current best sub-path and the location of each processor in the search space. This display allows the user to examine how well the search heuristics are guiding progress and where in the search space the processors are concentrating their search effort.

4.4.1.2 Search Effort Distribution History. The second display provides an indication of the distribution of the search effort. This display provides a two dimensional representation of the search space and uses color coding to indicate how often a column in the search space has been visited (Figure 13). While a true three dimensional representation would be superior, the

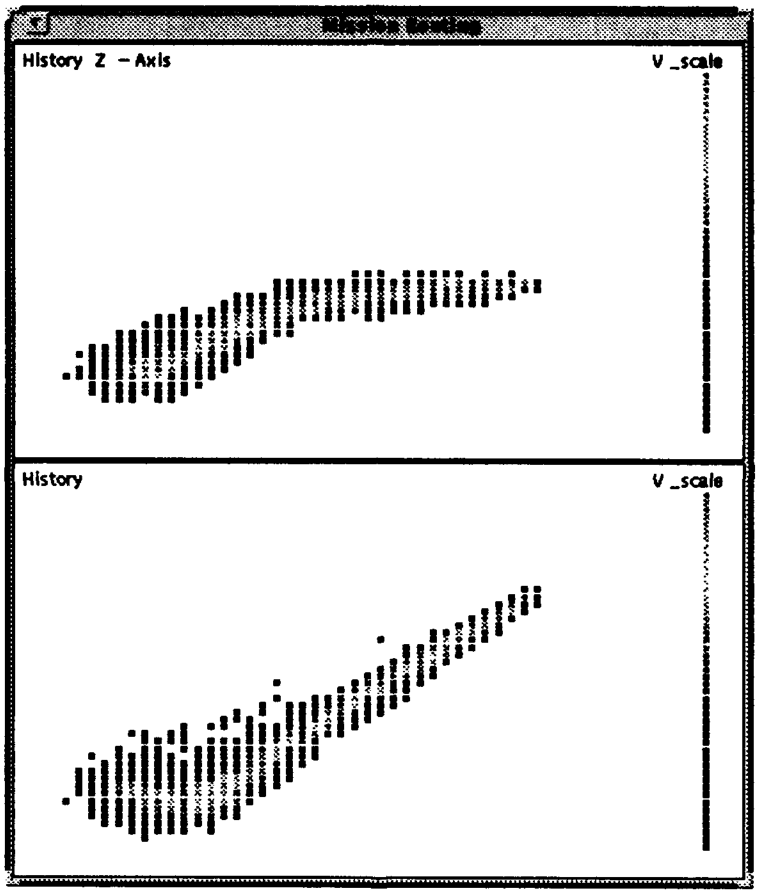


Figure 13. Two Dimensional Search Space Display

current animation speed would be significantly reduced. To compensate for this, AAARF was expanded to include the capacity to generate files suitable for input into the Matlab (41) or Explorer (75) for data visualization. An example of this representation is shown in Figure 14.

This display proved particularly useful in guiding the development of new search heuristics. It highlighted particular shortcomings that resulted from the problem domain. During the search the A* algorithm expands a frontier of equal cost. However, in the absence of any radar, the cost remains constant across a significant number of sub-paths. The original implementation provided only a weak upper bound on the path cost and thus the A* algorithm degenerated into a breadth first early in the search process. This was highlighted in the animation where the center of the search mass was distributed in the areas with no radar coverage. In fact, in the majority of problem cases, 90% of the search effort was directed to searching areas that were irrelevant to the final path. Paths which were 10 times the optimum distance were actually examined during the search including ones that moved in the opposite direction to the goal.

This observation lead to the development of guiding heuristics that significantly penalized partial solutions which did not reduce the distance to the target (14:6-7). Once this was corrected, further limitations were determined with the detection of paths that looped back to the destination. This further increased the search space and again triggered refinement of program search heuristics. This highlights the important of using visualization to ensure a better understanding of observed program behavior.

4.5 Development of User Interface

One advantage of the AAARF system is its ability to effectively manage interaction with the user. The interface obtains a system partition on the remote host, passes input parameters, and initiates execution of the mission routing software. For the mission routing program, the user can select from the control panel the following parameters:

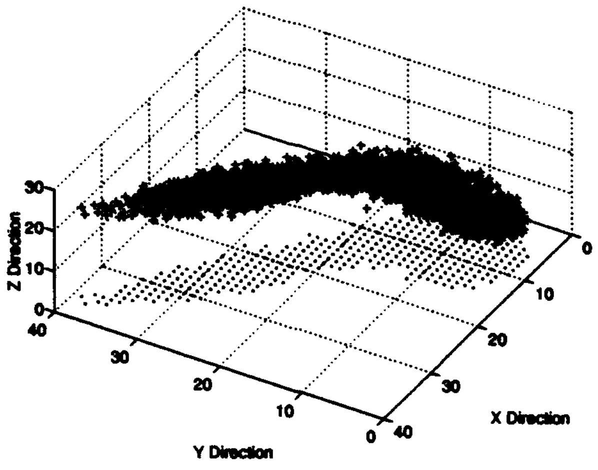


Figure 14. Three Dimensional Search Space Display

- Aircraft Route (ato file).
- Locations of radar sites and their performance.
- Terrain map.
- Number of processing elements allocated.

This particular feature is an important aspect of the AAARF implementation that is overlooked in other animation systems. In our particular application we are interested in allowing students to examine the execution of complex parallel programs executing on various computational platforms. Considering the lack of standardization of commands, operating systems and program development standards for parallel systems, the new user is often unable to even execute sample programs. The AAARF system is particularly helpful in this area and can abstract from the user complex input data requirements. Even the most experienced parallel programming students can lose weeks of research time determining idiosyncrasies of new machines. This type of interface support is not provided by any other animation system and appears to have strong merit in the educational environment.

4.6 *Summary of Results*

The instrumentation of the mission routing program enhanced the consideration of a number of important characteristics concerning parallel algorithm animation. When only the low level events were examined, it was possible to determine that the desired sequence of events were occurring, along with the effect of the underlying system hardware. From this it was possible to determine algorithm implementation efficiency. Without the addition of the higher level algorithm displays however, it is not possible to determine program effectiveness.

The difficulties in analyzing parallel programs, without some form of visualization, was highlighted by the fact that previous work had focused on optimizing the control structure to improve performance. While this would ensure greater scalability, the program was fundamentally flawed by one of its search bounding heuristics. Without including the AAARF animations in post execution analysis, algorithm development would have been significantly limited.

Further work in this area could focus on generating complete three dimensional animations of the search space. These representations could include terrain maps and radar coverage representations. The use of a specific data visualization tool would be required to allow adaptability.

The next chapter examines the use of application specific animations when applied to parallel discrete event simulation. Once again, generic animation displays provided very limited analysis capacity when applied to this class of algorithms, providing an excellent opportunity for application specific animations.

V. Animation of Parallel Discrete Event Simulation (PDES)

5.1 Introduction

This chapter examines the development of animation tools to support Parallel Discrete Event Simulation (PDES) (21). The majority of tools that currently exist for parallel algorithm animation focus on communication and execution tracing to provide the information for the users. For linear algebra and for many other application classes, the information about message traffic alone provides the majority of the information required by the user. For these problems, the mathematical definition provides the specification for the program. Once the decomposition of the data is known, the required operation on this distributed data dictates the communications. These displays are often complemented with tasks and data structure displays.

When visualization is applied to problems such as discrete event simulation, these animation techniques provide only a fraction of the information required for complete understanding. The chaotic message traffic patterns typically provide only limited insight into the performance of simulation. In addition, the interpretation of processor idle periods and other indicators of opportunities for optimization, cannot be directly related to particular phases of the simulations execution. This is further exacerbated since the traces produced are highly problem dependent and thus individual message patterns may not be truly representative of the program's execution. This section describes development of animations that address these problems. The utility of these animations is examined on both a VHDL (5) and Car-wash (83) simulations.

5.2 Parallel Discrete Event Simulation (PDES)

Discrete event simulation has long been used to create computer models of real world systems (21). The basic principle is that state changes in the system being modeled are considered to occur at discrete points in simulated time. The objects in the simulated environment react to stimuli by producing future events in simulated time. PDESs are simply the decomposition of large simulation models into a collection of smaller sub-models that can be executed concurrently (21). These sub-models are referred to as *logical processes* (LPs) and must interact with one another to enforce the real word constraints of the model. In the message passing programming paradigm, these interacting events take the form of messages between the LPs.

Ideal candidates for PDES simulation are systems whose *physical processes* (PPs) execute concurrently and can be modeled by message passing among their corresponding *logical processes* (LPs) (7:198–199). For example, electronic circuit systems can be simulated in this way, where the LPs representing the components, or groups of components, that make up the circuit are partitioned among the processors.

5.2.1 General Performance Model. The use of a global clock in distributed simulation constitutes a bottleneck because the LPs would all operate in lock-step. At any global time t, a number of LPs may have nothing to do. In *asynchronous* models, however, each LP contains a local virtual time (LVT), and the LPs are allowed to progress at irregular intervals. In most models, LPs communicate via time-stamped messages in the form of tuples, (t_k, m_k), where m_k is the message sent at LVT t_k (7:199). The specific rules for message passing depend on the particular protocol.

A global event-list would also be a bottleneck in distributed simulation. Therefore, each LP usually maintains its own event-list, or queue. Events either received or self-generated can be scheduled in the local event-list, if necessary, as well as sent to "downstream" LPs, as required by the model (7:198).

To ensure that the simulation reflects the real world problem, all events must generate the correct effect and these effects must occur in the correct order. This requirement is referred to as the *causality constraint* (21). That is, all events must be processed in time stamp order. Since the events are distributed across a number of processors in a PDES, the determination of which events can be processed is not a trivial task. The generation and consumption of events by the LPs is governed by the characteristic of the environment being modeled. As a result, the execution pattern is effectively non-deterministic due to the multiple cause and effect relationships being modeled.

5.2.2 Distributed Simulation Protocols. Synchronous simulation protocols can be loosely classified as either ***conservative*** or ***optimistic*** (63). Conservative protocols allow an LP to advance its LVT only when it is absolutely certain it cannot receive an event with a time-stamp less than the new LVT. The conservative implementation must include some mechanism to prevent deadlock (7). Optimistic protocols allow each LP to proceed at its own pace even though events may arrive out of the past. Optimistic techniques correct out of order messages by rolling back, i.e., restoring the state to a time prior to the actual message time and then recomputing forward. This rollback is invoked using some form of annihilation messages (63). The annihilation messages are passed between LPs to cancel events that were incorrectly generated. Within these broad classes there are an unlimited number of variations. Determination of the optimal protocol must often be decided by experimentation.

The animations examined here focuses on PDESs that utilize the conservative protocols. However, the concept could be extended to include the optimistic approaches, this is discussed in Section 5.6.

5.2.3 Deadlock Avoidance. Characteristics of the problem domain can lead to ***deadlock*** in the basic Chandy/Misra conservative simulation (7). This results from circular waits generated between dependent LPs for event messages. To avoid deadlock a number of techniques have been developed, deadlock avoidance using null messages, deadlock detection and recovery and synchronous algorithms (8, 63).

The simulations examined here use the NULL message approach to avoid deadlock. Under the NULL message approach events that contain lower bound time stamp information are distributed to dependent nodes down stream. However, this NULL message traffic has a significant impact on

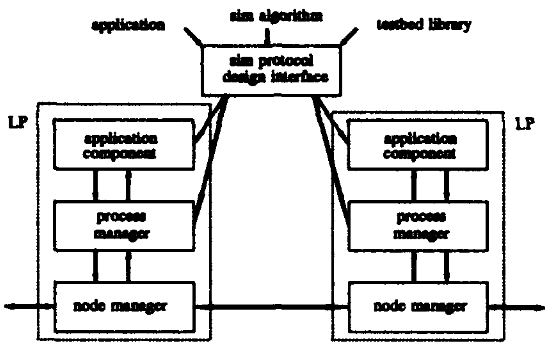


Figure 15. Block Diagram of the SPECTRUM Testbed from (64)

system performance (64, 77). To limit the impact of this NULL message traffic, many variations of the basic strategy have been employed that reduce the number of NULL messages (20).

The visualization proposed in this chapter highlights the characteristics of the NULL message traffic and allows analysis of NULL message reduction strategies. The animations are also useful in analyzing demand (20) and deadlock detection techniques (8). In these applications specific deadlock instances can be observed along with the protocol response.

5.3 Test Simulations

To examine the effectiveness of the proposed animations they were tested using two current AFIT research projects (5, 83). Both these simulations utilize the SPECTRUM simulation test bed (63).

5.3.1 Overview of SPECTRUM. In order to study *classes* of protocols for *classes* of applications Reynolds from the University of Virginia developed SPECTRUM (Simulation Protocol Evaluation on a Current Testbed using Reusable Modules) (65). SPECTRUM is a common testbed used for creating parallel simulations by taking an application and breaking it into application components, i.e., "pieces" of the application that run concurrently. Each application component, along with a process manager and node manager, make a logical process (LP), as shown in Figure 15.

The process manager provides *LP-level* functions to the application for initialization, local clock management, and event handling. The node manager provides hardware-specific functions to the process manager for event traffic among the LPs. To implement specific protocols, filters are written that "intercept" an LP-level function call by the application. The filters may then invoke protocol-specific actions, such as null message generation, LP *polling* for a message, etc.

AFIT has continued to maintain the SPECTRUM testbed as a baseline for simulations and has further modified it to meet AFIT's requirements. Spectrums application independence has assisted in simplifying experimentation with various message cancellation techniques. However, the major advantage for visualization purposes is the standardization of data collection. The selection of points of interest, and the subsequent insertion of the instrumentation code to record data structures, can be particularly time consuming. The use of SPECTRUM allows this instrumentation code to be inserted independent of the target application.

5.3.2 Simulations. The two applications that utilize SPECTRUM were examined using the visualization techniques described. The first is a simple queuing Car-wash model with feedback that consists of; 3 sources; 4 servers(washes); and 1 sink(exit) (83). In this model the cars can be re-washed and a percentage are feedback to the sources. This effectively models a number of interconnected servers, merge points, and consumers.

The second simulation examined consists of a circuit simulation program that is based on the VHDL description language. This parallel VHDL simulation system was implemented by Breeden (5). This simulation system decomposes the circuit behaviors into LPs which are then allocated to processor nodes. The results of these simulation runs are the actual traces of the circuit operating. The actual circuit examined by the simulation was a Wallace tree multiplier which consists of 1050 behaviors and can be decomposed into an arbitrary number of LPs. The design for this device was

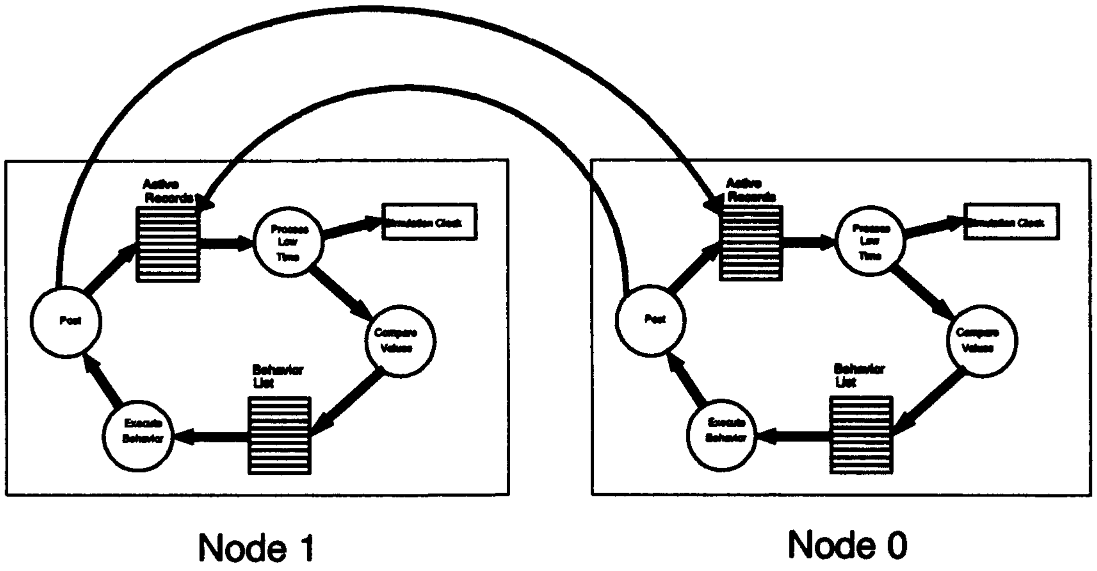


Figure 16. Two Node VHDL Simulation Model

taken from Hwang and Briggs (34). The circuit consists of a shifted multiplicand generator which generates intermediate results. The output of this is fed into a series of carry save adders, and then a carry propagate adder where the twelve bit product is generated.

Other circuits have been examined including an 8 bit carry lookahead adder, 32 bit bit/byte shifter, and ripple carry adders (5). A block diagram of the VHDL simulation system is shown in Figure 16 for a simple two node configuration.

The SPECTRUM test bed allows the instrumentation of only the service level routines and thus remains independent of the application and its configuration. Without this standardization the instrumentation of the programs under examination would become a significant burden. This task is simplified since AAARF simply redefines relevant Intel system calls, and thus modification of the code is not required for basic event types. To provide the additional displays, self defining interesting events are used to captured simulation times, event type, event times, event queue lengths, and SPECTRUM function calls.

5.3.3 Pseudo Real Time Displays. The most significant advantage of utilizing AAARF for animation of parallel discrete event simulations over other animations is the pseudo real time data collection. Considerable frustration has been encountered with debugging simulation systems. Even with relatively small circuit simulations, for example, the Wallace Tree Multiplier problem, execution times of 10 minutes have been recorded for various decompositions. These long execution times and remote hosting lead to considerable difficulty in program debugging. By utilizing various trace levels, constant monitoring of system progress is possible. It is also possible to determine the state of simulation prior to failing. This can be particularly beneficial for examining simulations that are prone to deadlock. While this low level of tracing obviously has a significant impact on execution time, the use of the node memory as trace buffers can minimize this overhead. While node buffering is limited by available memory, the continuous event dumping feature allows tracing, with reduced accuracy, of entire simulations that would otherwise not be possible. The tracing can later be enabled as required to examine only particular sections of the execution more accurately, once points of interest have been determined.

5.4 Animation Techniques

When visualizing parallel applications such as linear algebra operations and other mathematical functions progress can be readily determined from standard trace displays. This is a result of the deterministic nature of the operation being performed and its resultant algorithmic implementation. In many application areas the data structures manipulated by the program are contained in the message traffic; thus program phases can identified by this message traffic. For these types of applications the message traffic is relatively independent o rticular problem and directly reflects the algorithms design. Thus optimizing the message traffic based on a single observation can in many cases improve execution in general, for this class of problems. The data structures, and the content of the messages passed, represent the problem dependent information.

With PDESs the actual generation and consumption of messages is almost totally problem dependent. While the actual volume and transmission of messages is influenced by the simulation protocol used, the actual specific problem remains the dominate factor. Thus observing the communication of real and null messages simply reflects the occurrence of events at particular LPs. Any information gained from sequencing, or individual observation, is unlikely to be of use in general analysis of the PDES. Progress of the PDES can only really be determined by examining the local virtual time LVT at the each LP.

While lower level processor activity displays provides information about work done by the processors, the value of this activity remains unclear. For example processors can continue to consume null messages. However, if simulation time progress is not made, then *effective* work has not been achieved. Thus to provide useful animations of discrete event simulations it is important to combine low level tracing of communication events along with the data structure they contain. The low level animations allow us to generate the standard utilization and message traffic animation. By combining the result of thes vel events, trends in channel usage, transportation delay, processor activity, and other important characteristics can also be examined.

The higher level information provided by instrumenting simulation clocks and other data structures allows clearer examination of the simulation process. Depictions of these data structures provide the only meaningful method of examining the progress made by the simulation. The detection of deadlock, violation of causality constraints and other program abnormalities can be determined from this information. If the animations were simply depicting the basic message traffic and processor utilization, these abnormalities would be difficult to determine.

Thus the goals of the PDES animations are:

- Determine system mapping characteristics.
- Depict simulation time progress.
- Highlight individual LP execution.
- Determine true processor utilization.
- Depict individual simulation phases.

Based on these animation goals the following parameters are selected for recording by the instrumentation system.

- Simulation time at each LP.
- Latest time stamp message sent by an LP.
- Message type (Real/Null).
- Simulation time of each message.
- Channel delay for each message.
- Receive blocking times for each message.
- Message queue length at each LP.
- Time of function calls and returns (Function Profiling).

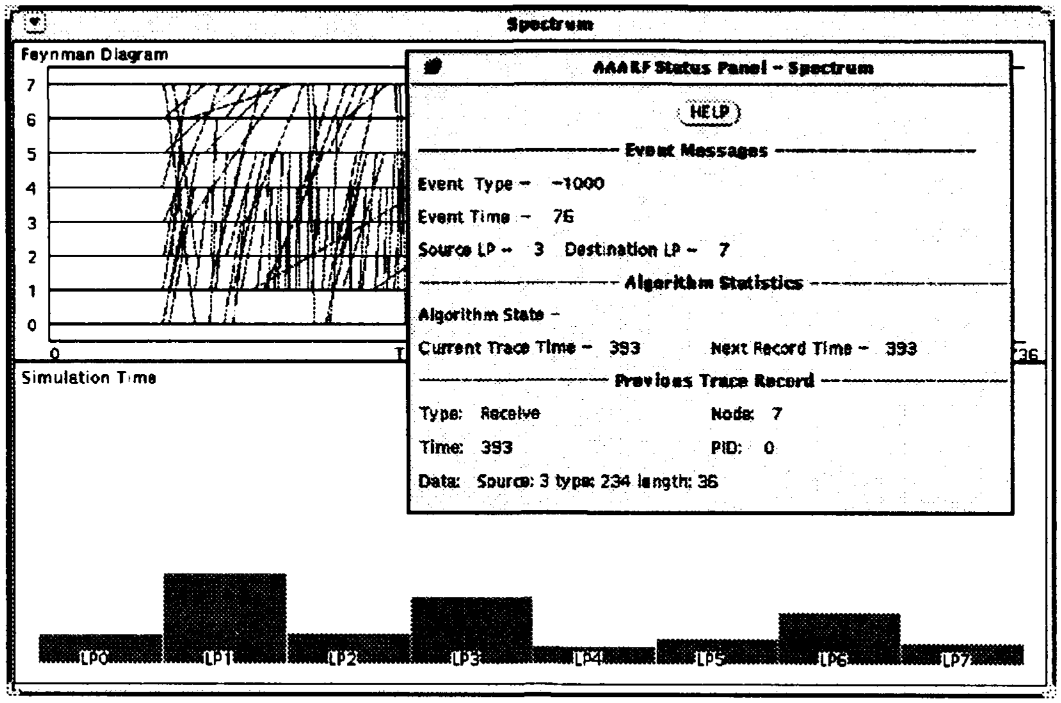
While this information can be readily collected, the volume of data recorded over an execution cycle can only be effectively assimilated using graphical techniques. The animations that have been developed are motivated by two related uses, education and program/algorithm development.

- From an educational perspective the animations aim to explain the operation of PDESs and re-enforce educational objectives.
- From the program/algorithm development perspective the animations are aimed at providing a tool to be utilized by the more experienced user to assist with debugging and optimization of PDESs.

The following section outlines the operation and use of the displays that have been developed. A total of nine displays have been created or adapted to specifically animate the important characteristics of PDESs. The displays developed are:

- *Simulation Time Display:* Depicts simulation time progress on distributed LPs.
- *LP Matrix Display:* Depicts LP interdependencies.
- *Null/Real Message Display:* Reveal synchronization protocol overhead.
- *Progress Display:* Highlights simulation time progress and message volumes relative to execution phases.
- *Message Queue Sizes:* Depicts internal program queues information.
- *Channel Statistics:* Depicts message traffic load.
- *Message processing time:* Depicts computational performance.
- *Blocking Statistics:* Accounts for busy wait times to provide accurate utilization information.
- *Lookahead Display:* Depicts simulation protocol performance.

5.4.1 Simulation Time Display. This display provides a simple bar graph representation of the simulation time at each node. It provides clear indication of how simulation time is progressing at each LP. Its main use is in detecting LPs that are making poor time progress. It scales well to 512 LPs. An example of the display is shown in Figure 17 for a 8 LP Car-wash simulation. Figure 17 also shows the conventional Feynman diagram and the simulation status panel. Note the complex inter-node communication patterns. The status panel provides more detail about instantaneous events such as message type and time stamp information. In this case the free running LP1 is making significant progress relative to the other LPs. On larger simulations the problem type and the LPs look-ahead ability may combine to produce an unexpected progress by LPs. While mainly for educational purposes, the display does highlight relative progress at each node. In particular it can highlight LP that make infrequent time progress.



AAARF Status Panel – Spectrum

Event Messages

Event Type – -1000

Event Time – 76

Source LP – 3 Destination LP – 7

Algorithm Statistics

Algorithm State –

Current Trace Time – 393 Next Record Time – 393

Previous Trace Record

Type: Receive Node: 7

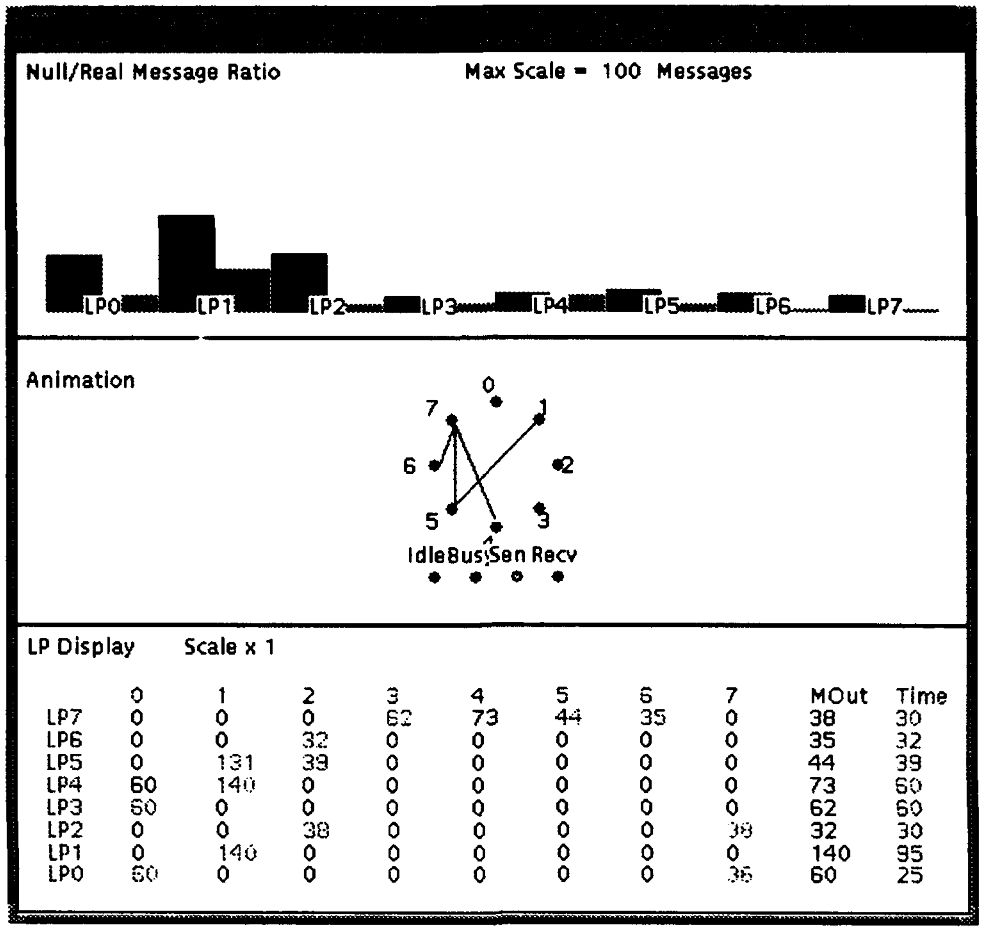
Time: 393 PID: 0

Data: Source: 3 type: 234 length: 36

Figure 17. Simulation Time Display for Car-wash Simulation

5.4.2 LP Matrix Display. For small simulations, limited by screen resolution and the user's interpretation ability. it is possible to examine each LP's performance in more detail. The LP matrix display depicts the interconnection between various LPs in a matrix form and shows the latest message times on each arc. Color coding is used to depict null and real messages and the lowest and highest time stamp messages in the simulation. This display is shown in Figure 18 for a circuit simulation. 'MOut' reflects the latest time stamp message sent by a particular LP, and 'Time' indicates the LVT at that LP. In this example LP3 has a LVT of 60 units, based on receiving a null message from LP0 at time 60 units, the only LP on which it is dependent. Based on it's minimum processing time of 2 units it has sent a null to the only LP7, the only LP dependent on its output.

5.4.3 Null/Real Message Display. When evaluating different null message cancellation techniques. it is important to examine the volume of null and real messages produced by the



LP Display Scale x 1

	0	1	2	3	4	5	6	7	MOut	Time
LP7	0	0	0	62	73	44	35	0	38	30
LP6	0	0	32	0	0	0	0	0	35	32
LP5	0	131	39	0	0	0	0	0	44	39
LP4	60	140	0	0	0	0	0	0	73	60
LP3	60	0	0	0	0	0	0	0	62	60
LP2	0	0	38	0	0	0	0	38	32	30
LP1	0	140	0	0	0	0	0	0	140	95
LP0	60	0	0	0	0	0	0	35	60	25

Figure 18. LP Matrix , Null/Real, and Animation Display

simulations. This display depicts this information for each LP and updates with each message sent. Much of the simulation research conducted at AFIT focuses on the development of tools to automate the partitioning of behaviors into the LPs. This process is often analyzed using volume of nulls and reals generated as one performance metric.

5.4.4 Progress Display. The progress display combines a number of interesting system parameters and depicts them relative to the execution time for simulation. By selecting an execution time interval, the following parameters are displayed in a scrolling bar graph for each time interval.

- Average Simulation time progress.

- Message volume nulls/reals.
- Max and min simulation time progress.

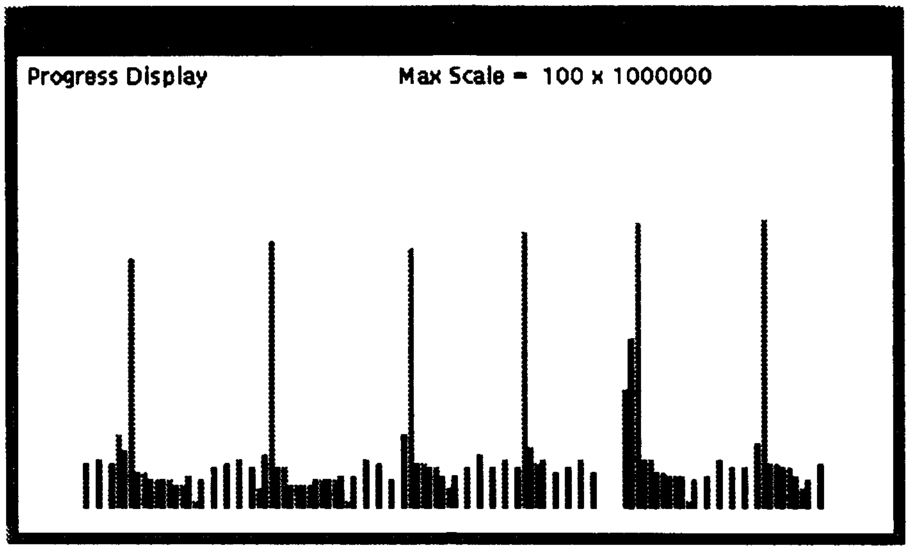


Figure 19. Progress Display Simulation Time Progress

This display is shown in Figure 19 and has been useful in focusing attention on periods of poor simulation progress. It has also been beneficial when examining the simulations to determine which LPs and event and important to simulation progress. For example, determining which dependences are actually delaying execution. This information can help guide LP to node allocations. Figure 19 shows the progress by the VHDL Wallace tree multiplier simulation. In this case the lighter bars indicate the progress made by the lowest LVT LP and the darker bars the highest LVT LP.

Figure 20 shows this same display depicting the volume of null and real messages transferred per unit time. The lighter bar indicates the null messages and the darker bar the real messages. Summing two adjacent values, nulls first, gives the total number of messages transferred in the sample period.

The horizontal axis represents program execution time and the sample period is selected on the operators panel. On both of these figures each sample point represents a 100 ms period.

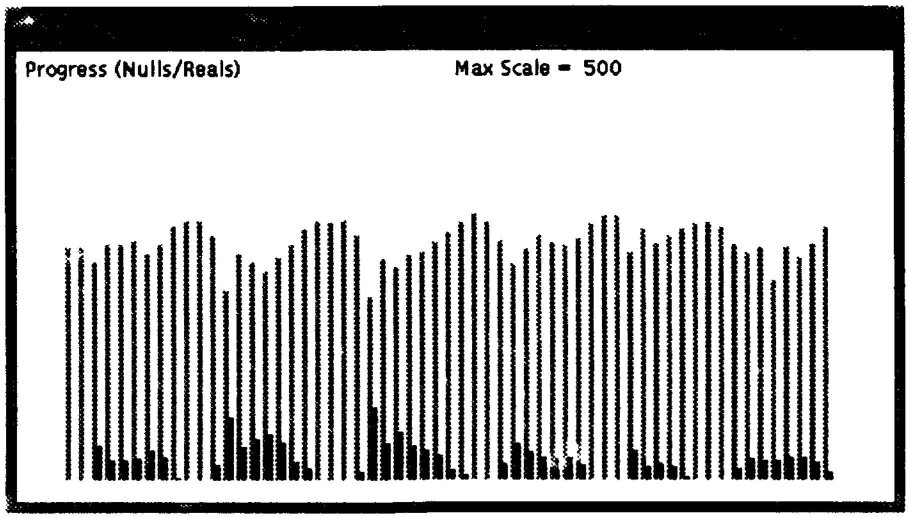


Figure 20. Progress Display Null/Reals Volume

This animation can be particularly useful when developing NULL message reduction filters, and deadlock and recovery strategies. AAARF allows analysis both algorithm and system state, in pseudo real time, to be examined at the point of deadlock. This Information can significantly assist with program development.

One interesting use of this animation is in examining the communications capacity of the target processing systems. For any given architecture communications between nodes is restricted by the interconnection network, transportation delay, and link bandwidth. Thus for a given system there is a maximum number of messages that can be routed through the system in a given time period. While this volume varies with computational load, simulation protocol, LP decomposition, and other parameters, empirical measurements are possible. Our experiments have indicated that for a particular application program the message volume remains relatively constant, for a problem class and a fixed number of node processors. From these experimental results and message volume estimates for a particular problem it is possible in some cases to estimate program performance.

5.4.5 Message Queues Size. By displaying the length of message queues we can gain an understanding of the backlog of events currently at each LP. The message queue display shows this information along with the maximum size reached by each event queue during the simulation. The presentation of this information is similar in format to the simulation time display.

5.4.6 Channel Statistics. This display provides information about message traffic between processor nodes in addition to the standard link volume displays provided by the basic AAARF system. While the message volume per link is important, this animation provides an additional graphical view of the inter node transportation delay. It aims to portray to the users the effects on inter-node communication times due to message volume. For example, on the iPSC/860, the time required to send a single byte message between nodes is approximately 1ms on a unloaded system. However, this is simply a lower bound since the effects of channel traffic and the receiver activity can greatly increase this figure.

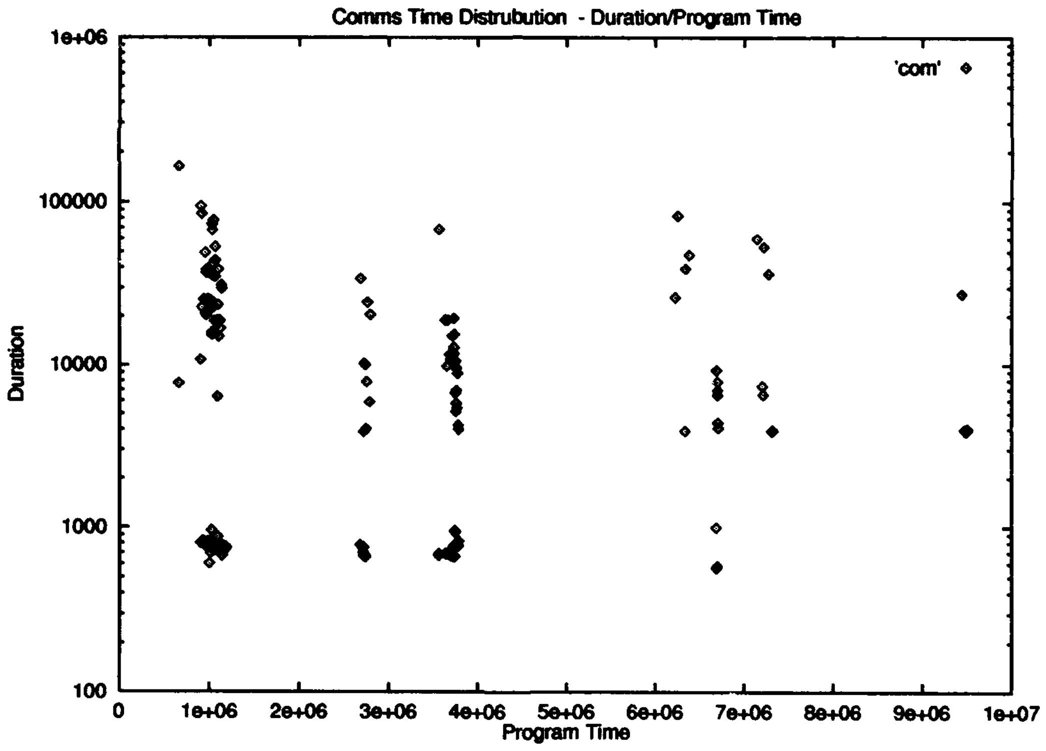


Figure 21. Channel Statistics (File Output)

AAARF provides the transportation delay information in two formats. The first is simply a text file that is suitable for a plotting package such as gnuplot©. This representation is show in Figure 21. The horizontal axis in this diagram is program execution time. The vertical axis represents the message transportation time, with the actual events marked relative to the sending time.

The second is an animation of the distribution of these message transportation delays. In this view the transportation delay is divided into five user defined categories. Within each category the number of message that encountered that delay are shown for each processor. In Figure 22 eight processing elements are shown for the VHDL simulation of the Wallace multiplier. In this example we have significant transportation delays which could result from receiving nodes executing the receive command late, or channel conflict. However, in this case the receiving nodes are spending a considerable time blocking for messages (see Figure 23). This indicates considerable channel conflict, which is in fact the case in this random partition of the Wallace tree multiplier. This animation allows multiple configurations that reveal particular aspects of the programs execution.

- Depiction of all inter-node communications or any subsets of nodes.
- Include or remove the contribution due to receiving node not ready.

This display can be used to help determine communication channels that are producing significant delays. For example, the distribution of message transportation times when the receiving node was already waiting (blocking) highlights channel conflicts. Alternatively, displaying only the message transportation time for messages where the receiving node did not wait reflects the processor utilization. The overall distribution of transportation times , in general, provides a relative performance measure for the program.

5.4.7 Message Processing Time. This display shows the distribution of computation time required to process an incoming event. A simple scrolling bar graph animation is used to display this data. The display also includes a static bar that shows the maximum, minimum, and average event processing times. This display gives a clear indication of the computation load generated by events at a particular LP.

5.4.8 Blocking Statistics. This display shows the distribution of receive blocking and send blocking times experienced by a particular LP. In many animations systems including AAARF the processor is considered idle when blocked for communications. This idle/active representation is then used to generate the utilization displays. This relies on the program under examination only using blocking communication primitives. However, in the case where an LP simply probes for incoming events, before executing the receive instruction, the utilization display becomes meaningless. To overcome this problem, trace data is collected that records calls made to functions that probe the message queues. Communication blocking is considered to commence at this point. Thus, this display gives a clearer indication of the time spent waiting for events by each LP. The display can be configured to display both the send blocking and receive blocking times.

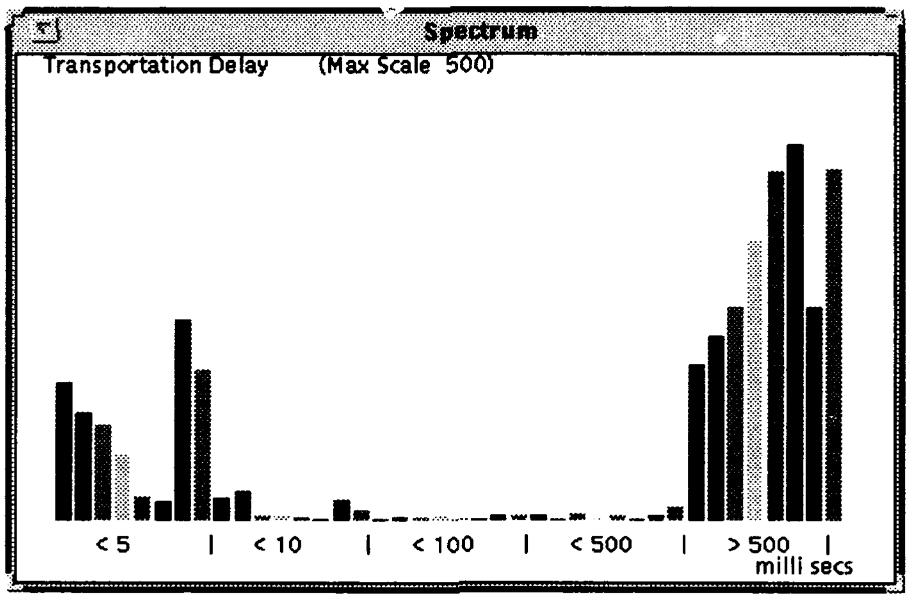


Figure 22. Channel Statistics Display (Transportation Delay)

When combined with the channel statistics display it can be used to determine processor utilization and communications channel loads. An example of this display is shown in Figure 23.

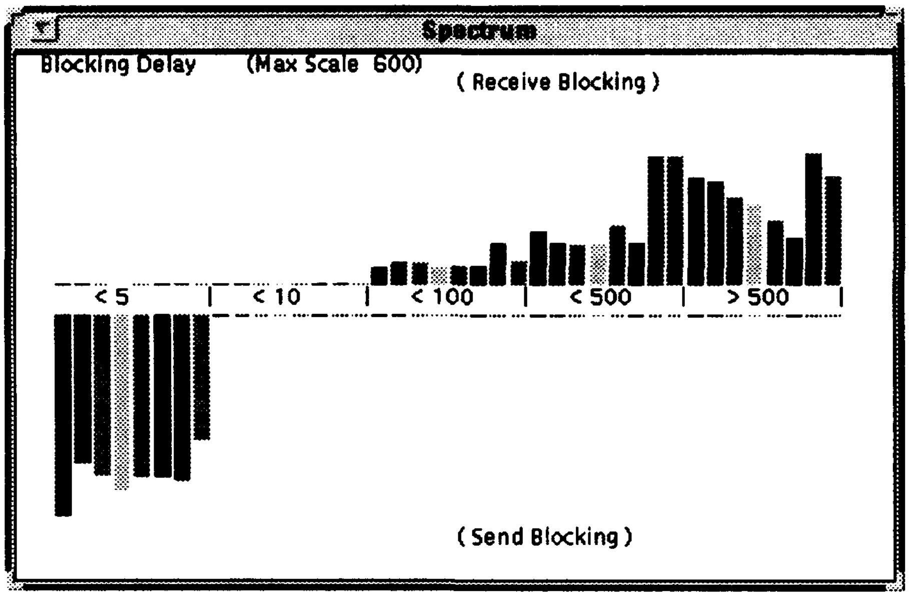


Figure 23. Blocking Statistics Display

Long periods of blocking for events indicates an LP is unable to continue effective work. While this is inevitable in many simulations, excessive blocking can highlight areas where LPs can be co-located to improve simulation performance. The converse is also possible for computationally bound LPs. Computationally bound LPs will be reflected by nodes that have low blocking times and large events queues. In this case an alternative partitioning of the problem between LP may be warranted.

5.4.9 Lookahead Display. The ability for LPs to predict the future based on their own characteristics and knowledge of the past is referred to as lookahead. The ability of the program to exploit lookahead is critical to achieving good conservative simulation performance. The lookahead animation shows the amount of lookahead generated by the individual LPs. This is shown as a deviation from the current LP simulation time.

5.4.10 Phase/Task Display. For the simulation to make time progress, each LP must obtain, process, and generate new event messages. This cycle of execution includes insertion of events into queues, actual computation of PP behavior, and communication with other LPs. The time required to perform these and other operations is dependent on the combined behavior of all LPs. For example, LPs may be idle waiting for later time stamped messages from an upstream LP. Thus, the time required to obtain the next event can vary greatly during execution. To better understand the execution time required by each of these phases a variation of the AAARF Task display is utilized. The modular design of SPECTRUM allows the instrumentation system to record each function call made by the LP to the SPECTRUM simulation services. By animating these function calls we can highlight particular aspects of the program's execution.

For example, Figure 24 shows the time required by LPs to send and receive events between nodes. In this diagram the dark regions show the periods during which an LP posts a message to an LP on another processor, and the lighter area shows the time spent by an LP waiting for incoming messages. The white sections in this trace represent execution of functions unrelated to these two activities.

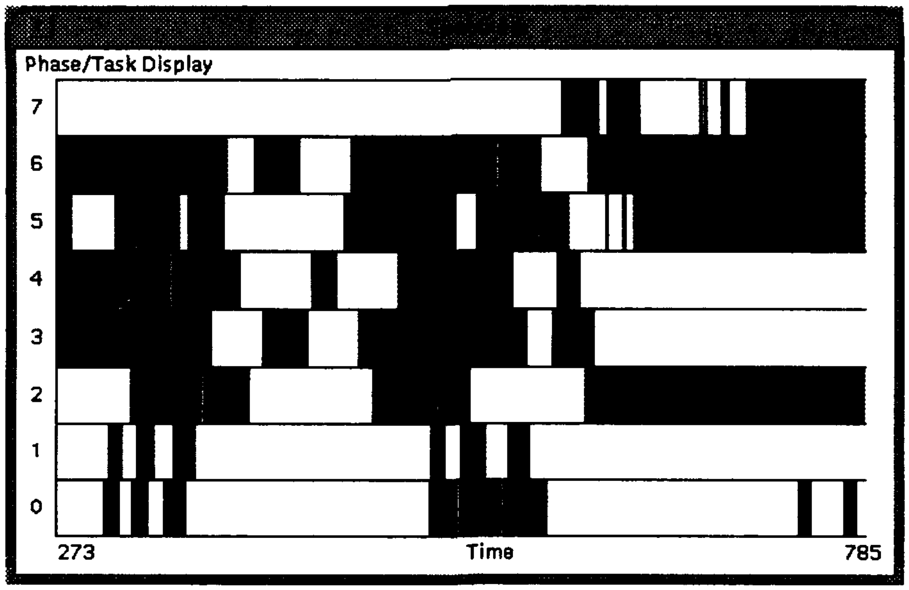


Figure 24. Phase/Task Animation Display

The display has been implemented to allow these functions to be grouped and color coded in any manner desired by the user. This allows any particular combination of parameters to be examined. For example, all functions that manipulate event queues can be assign a particular color and contrasted against all other functions.

It is also possible to obtain a clearer representation of system utilization. This display can be used to depict the processing of real events as effective LP utilization, other activities as overhead, and any send and receive blocking as idle. In this configuration the display can give a clear indication of the amount of concurrent execution achieved by the simulation program.

5.5 Execution Analysis

The animation of the program's execution provides a more complete understanding of both the algorithm and its mapping to an architecture. This is important from an educational perspective, as is the ability also use this information to optimize program performance. There are three main areas in which the simulation displays can be used to optimize performance.

- Architecture mapping.
- Message volume minimization.
- Task Scheduling and Load balancing.

Optimization of parallel discrete event simulations is a particularly problem dependent process. Even within a class of similar problems small variations in the problem space can result in significant variations in both the execution pattern and simulation run time. Visualization and its use in optimization is unlikely to completely overcome this problem. However, it is a useful tool with which to examine these variations and assist in evaluating and locating possible areas for improvement.

5.5.1 Architecture Mapping. Partitioning and mapping of discrete event simulations to particular architectures is an optimization problem that is known to be NP-hard (22). As a result a number of heuristic approaches have been employed to find solutions that are close to optimal

(70, 74, 78). The use of static graph-based partitioning techniques (10) can provide an initial solution, but these techniques cannot adequately address the non-deterministic execution pattern of the simulation LPs. It is impractical to determine the actual order of execution of various LPs prior to execution of the simulation, since this effectively amounts to the solution of the problem being simulated. The message traffic required between two particular nodes in an LP dependency graph can vary greatly between actual effective executions of the LPs, i.e ones that process real events. As a result, efforts to apriori schedule LP executions is also thwarted.

This problem has resulted in many simulations requiring experimentation to determine effective mapping for particular configurations. The general approach is to produce LPs from the basic physical processes PPs that result in the minimization of dependencies between LPs. Once this has been accomplished the LP must be mapped to the physical processors. This mapping aims at minimizing communication delays and channel conflicts between dependent LPs while evenly distributing the workload. Examples of this approach are discussed in (10).

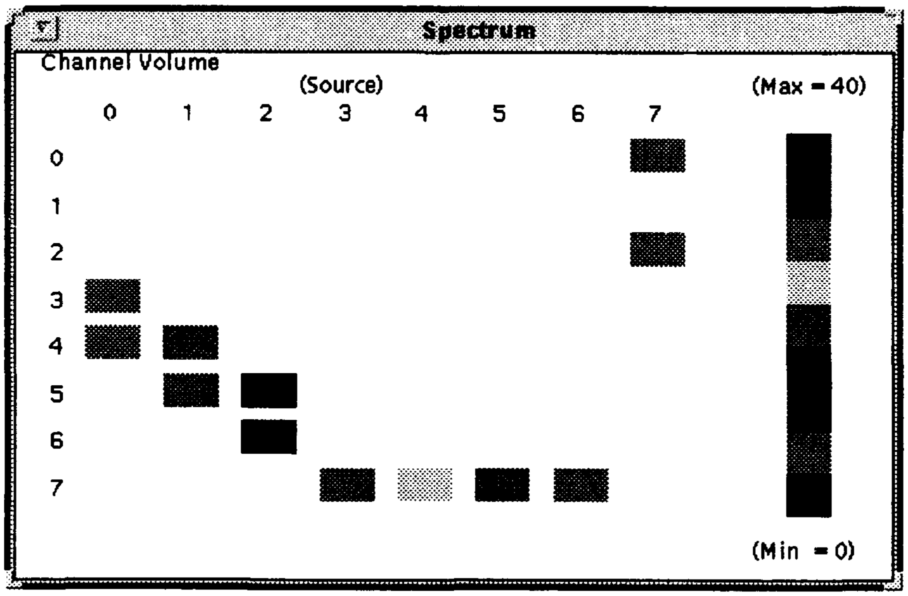


Figure 25. Channel Volume Display (Virtual Channels)

However, this mapping is based on static dependency data. The actual dependencies created during execution vary as a result of the progress made by different LPs. For example, link message traffic is dependent on signal activity in VHDL simulations. Where signal activity is defined as the number of events generated during the simulation on this signal. Since there is no information available on signal activity, partitioning can only be based on structural information of the circuits to decide which signals and elements will be assigned to which partition. Thus once an initial mapping has been achieved variations can then be examined based on experimental data. The animation such as the Phase/Task display and Channel Statistics display can be used to focus the optimization process on specific problem areas.

For any reasonable sized parallel implementation a fully connected processor system is impractical. As a result, the minimization of communication channel conflicts can significantly improve performance. In some applications the critical path can, to some extent, be determined and communication traffic arranged to minimize this conflict. This is typically achieved in linear algebra application programs and can be finely tuned by experimentation (28). In these applications the sequence and volume of communications can be determined and this information incorporated into the mapping strategy. This would normally be achieved by weighting the edges in a dependency graph of the LPs.

The information provided by the animation can assist in both the analysis and optimization of these mappings. By combining the AAARF communications volume display information, shown in Figure 25, with the LP dependency graph, bottlenecks in communication channels can be detected. LPs must often share communication channels as a result of a particular mapping. These shared channels can be examined to determine if the average message delay time is significantly greater than the systems minimum. If actual execution results indicate that a particular link has a significantly greater than average usage and delay this information can guide the development of alternative mappings. Care must be taken to ensure this effect is not due to the receiving node executing the receive command late, due to its computation load. This can be checked by examining the Blocking

Statistics display to ensure the node is regularly blocking for incoming messages. If this is not the case, the extended communication time indicates a load balance problem in the receiving node.

If these LPs are also significantly trailing the other LPs in both current simulation time at the LPs and the highest time stamp message sent (as can be shown on the LP matrix display) they are likely to constitute a section of the critical path. This process focuses optimization efforts on sections of the critical path that were previously unclear.

5.5.2 Load Balancing and Task Scheduling. The simulation displays can be used to assist in the task of load balancing by being configured to highlight nodes/LPs that:

- are consistently waiting to receive input messages, and
- have simulation times that are consistently higher than the lowest current simulation time on any LP.

This information can be taken directly from the LP Simulation time and Node Blocking Statistics displays. A peak on both of these displays indicates that a node has successfully completed processing available event messages and is not delaying progress of the simulation. Underutilized processors are reflect by LPs that have a consistently empty message queue and a peak on the node blocking statistics display. They indicate possible opportunities for the allocation of additional LPs or computationally intensive ones.

For simulations with static LP allocation, our ability to produce a well balanced implementation, is directly related to the generation of task schedules. The non deterministic execution pattern of LPs and variations in task execution times, significantly limits the generation of task schedules. While this remains a difficult problem to overcome, a number of techniques have been examined and are discussed in (70, 74, 10). For pedagogical examples it is possible to use the information generated by the phase task and message processing time displays to confirm applicability of task schedules. This is particularly useful in demonstrating scheduling algorithms in the educational environment.

5.5.3 Minimization of Communication Volume . The simulations examined by this chapter include various null message cancellation strategies that assist in minimizing the volume of null messages transmitted between LPs. While these messages are essential (in conservative PDESs) to avoid deadlock, they are pure overhead and consume vital communication bandwidth. A number of displays have been created to show various null/real message counts both during and after execution of the simulation. While this helps with optimizing the mapping and the selection cancellation technique, further improvement is possible. As with free running sources in the Car-wash simulation, LPs making better than average simulation time progress can tend to saturate down stream communication channels. In these cases communication channel transportation times increase due to channel conflict. Low simulation time LP are then further delayed while waiting for events on incoming dependent arcs that share these communication channels. The result being that simulation progress is reduced due by the high volume of later time stamped events in the communication network.

By examining the LP matrix display, the user can easily detect LPs that are sending messages that are significantly later than those currently being processed by the receiving node. The matrix display can assist in determining strategies that constrain the progress of some LPs to give greater over all time progress. Care must be taken to ensure that the beneficial effects of lookahead are not subsequently constrained. That is, the later time stamped messages are simply waiting in the event queues of down stream nodes. The periodic effects of this problem can also be examined by using the progress displays.

5.6 Expansion to Support Optimistic protocols

While this research has focused on animations for conservative simulation protocols expansion to include optimistic approaches is possible. To depict optimistic protocols, time related displays would need the ability to depict negative time progress. The displaying of additional message type statistics, i.e. for annihilation messages, would be required. Information dealing with roll back and storage of transient state information would also be required. Under the optimistic protocol we would still be interested in examining the distribution of work across the processors and the usage of physical communication channels. Thus many of the displays shown in later sections are applicable.

In the case of the time progress displays, rollback information would be the focus of analysis. Both the computational cost of this rollback and effectiveness of optimistic execution would be depicted. The progress animations would be also depict negative time progress. Statistics gather relating to NULL messages traffic could be replaced by annihilation message information. In this case showing their propagation and computational effect.

The displays depicting simulation overhead would focus on the storing and reloading of state information. The simulation time progress made at each node would tend to have a greater deviation from a global mean simulation time, than in the case of the conservative approach. This results from the conservative implementation constraining individual LP by a minimum time stamp increment. By examining the simulation time progress during execution, we obtain a better understanding of how optimistic execution is aiding simulation progress. This could be useful in examining varying degrees of thrashing that can occur in optimistic simulations (20). Events can arrivr out of order in optimistic simulations. Thus displays depicting the average and maximum rollback times encountered during the simulation, would be of interest in determining techniques to limit the unnecessary future computation.

The current lack of research into optimistic simulation techniques at AFIT limits the immediate applicability of this work, however future events may change this.

5.7 Summary

The animation techniques described in this chapter have provide useful insight into the execution of parallel discrete event simulations. They are a logical extension of standard parallel visualization techniques and highlight the benefits of complementing standard displays with application specific animations. The animations are a practical application-specific display class, since simulation development is often based on a set of underlying simulation primitives. This ensures that code changes required by the instrumentation system are reused.

While useful from both the educational and program development perspective, the displays can play an important role in optimizing the execution of the simulation program. Problem dependent execution patterns generated by PDESs reduce the user's ability to isolate and optimizing particular phases of a program's execution. However, the general trend information provided by the displays can assist with load balancing and problem decomposition. By detecting uncharacteristic system behavior, program optimization efforts can be more clearly focused.

The depiction of tend information was often well suited to post execution analysis displays. This resulted in the development of a trace conversion and analysis tool for the AAARF system. This enhancement is discussed in the next chapter.

VI. Conversion and Analysis Tools

6.1 Introduction

This chapter examines the development of data analysis tools that can be used to analyze trace data from a variety of application programs. The analysis tools highlight significant characteristics of a program's execution. These characteristics provide a clear focus for optimization of both problem decomposition, architectural mapping, and resource balancing. To enhance AAARFs compatibility with other systems, a data conversion facility that provides limited translation between PICL (23) and PRASE (42) trace data formats is also implemented.

6.2 Overview of Analysis Process

The time period between interesting events captured by the instrumentation system is often in the order of micro seconds (42), the actual duration being determined by both the processor's clock speed and the algorithm under examination. As a result, the instrumentation system can generate thousands of events per second of program execution time. Thus, when analyzing the execution of these programs on a Sun workstation, the replay time can be a two orders of magnitude longer than the actual execution time of the program. This is particularly limiting for programs that have long execution times, for example Mission Routing (14). This problem also leads to difficulties when analyzing programs that solve irregular problems. The execution patterns of irregular problems are determined by characteristics of the specific problem being solved. An example would be a search process where the control flow is altered by individual evaluations during execution. A regular problem such as matrix multiplication will have the same execution pattern irrespective of the particular problem under examination. Thus, it is possible to observe sections of the program's execution, when dealing with regular problems, to determine overall load balancing and communication characteristics. In the case of the irregular problems, transient behavior may not truly represent overall execution behavior, since load balancing and communication characteristics vary during execution. In these cases, a post execution summary of overall performance data is required.

It is possible to speed the animation by increasing the time step taken by the display system between updates of the display. However, this does not completely eliminate the problem since the events must still be examined, if not displayed, to ensure that the information present in the next screen update is accurate. This large time step can also result in interesting sections of the program's execution being missed. As a result of this, the animation system forces the user to closely watch the execution of the entire program for points of interest.

From this observation it became apparent that users require tools that allow the trace data to be quickly analyzed when collected off-line. This allows the user to focus on periods of the execution that contain events of interest. In addition, the original AAARF implementation focused on run time animation displays. These displays, while suited to their intended purpose, do not provide execution summary information. The only animations that began to tackle the problem of characterizing the whole program's execution, were the Kaviat and RVA animations.

6.2.1 Analysis Requirements. The aim of the additional analysis tools that have been developed are to present information about the complete execution of the program. In this manner they allow the user to determine periods of execution which warrant closer examination. For systems with an extensive number of nodes, the determination of subsets of these nodes that have interesting behavior is also important. For processing systems containing more than 256 nodes (39) animating the behavior of all nodes simultaneously is restricted by the display resolution. Thus, for larger systems it becomes important to provide displays that quickly localize attention so more detailed animations of small numbers of nodes or clusters of nodes can be animated.

To overcome this problem, the following basic system parameters were considered essential in determining overall program execution characteristics.

- Distribution in time of message transportation delays.
- Communication message volume between node processors.
- Communication message volume on physical processor interconnections.
- Node blocking time characteristics.
- Concurrency time profile.
- Overall processor utilization.

While this basic information is partially presented in a number of current AAARF animations, it must be extracted from the animations by directly comparing the traces for different node processors. The new analysis tool allows complete execution traces to be analyzed and the results summarized in a single display.

The distinction between logical and physical processor interconnections is important for analysis of mapping strategies. The partitioning of a program determines the volume of message traffic between logical program components. These logical components must then be mapped to the physical processing elements. While a number of mapping strategies have been developed, the message volumes can often not be determined apriority. As a result, it is important to examine how close the expected message volumes were to that message during execution. The combined effect of these variations can then be seen in terms of the loads on physical processor links.

6.3 System Design

AAARF was originally designed to provide a basic algorithm animation framework that can be expanded to include additional animation problem classes. These take the form of independent applications that are controlled by the AAARF base frame via socket communication. While this has worked successfully and continues to promote expansion, it has some limitations. The limitations result from the design decision to associate all active displays with a single application class. Thus, while within an application class multiple displays can be active, they depict the execution of only a single application program.

The additional displays are aimed at post execution analysis of processor utilization and message passing activity. They must provide the capability to analyze results from a single execution and allow comparison with other results. It is also important that the tools be available to analyze the current program and depict data gathered from previous executions. For this reason, the additional displays were developed as separate tools that can be executed from within any display class. Thus, the displays can depict results obtained from other executions, while the current display class depicts the results from another program or implementation.

A system structure diagram of the additional analysis tool is shown in Figure 26. It is implemented as a separate compilable application, as are all other display classes. It can be compiled independently of AAARF, if required, and can be selected for execution from the AAARF base frame menu. Note that all AAARF applications are designed to produce independent executable files. This approach eliminates coupling between different developer's display classes, leading to a more robust design. The implementation follows the same basic structure of AAARF with separate files for each display type. The analysis tool includes a control panel which is used to initiate one of the trace file conversion programs. After completion of the file conversion programs, the control panel *main* routine in turn *forks* the relevant display class.

The control panel passes the filename to the conversion program along with other user specified information. Each conversion type is contained as a separate function. The output of the conversion program is written to a specified file and in turn read by the display program. The output from the analysis tool consists of sets of (x,y) co-ordinate pairs. Note: An additional (z) co-ordinate is also added in some cases, for example mission routing. The file format for this output places each tuple on a separate line in an ASCII file. This text format is suitable for input into a number of plotting and data analysis packages such as, Matlab (41), Explorer (75), and Gnuplot

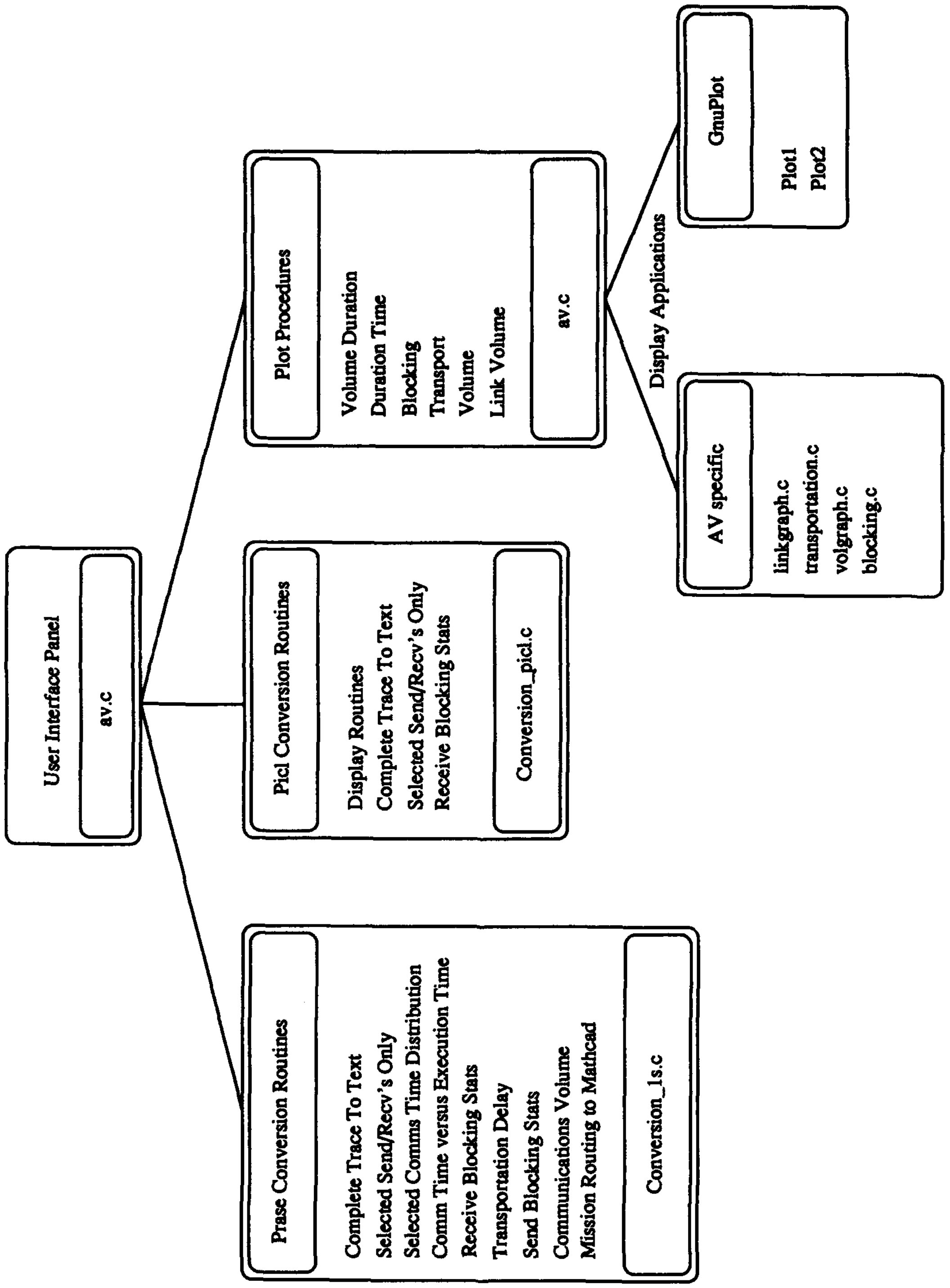


Figure 26. System Structure Diagram for the Analysis Tool

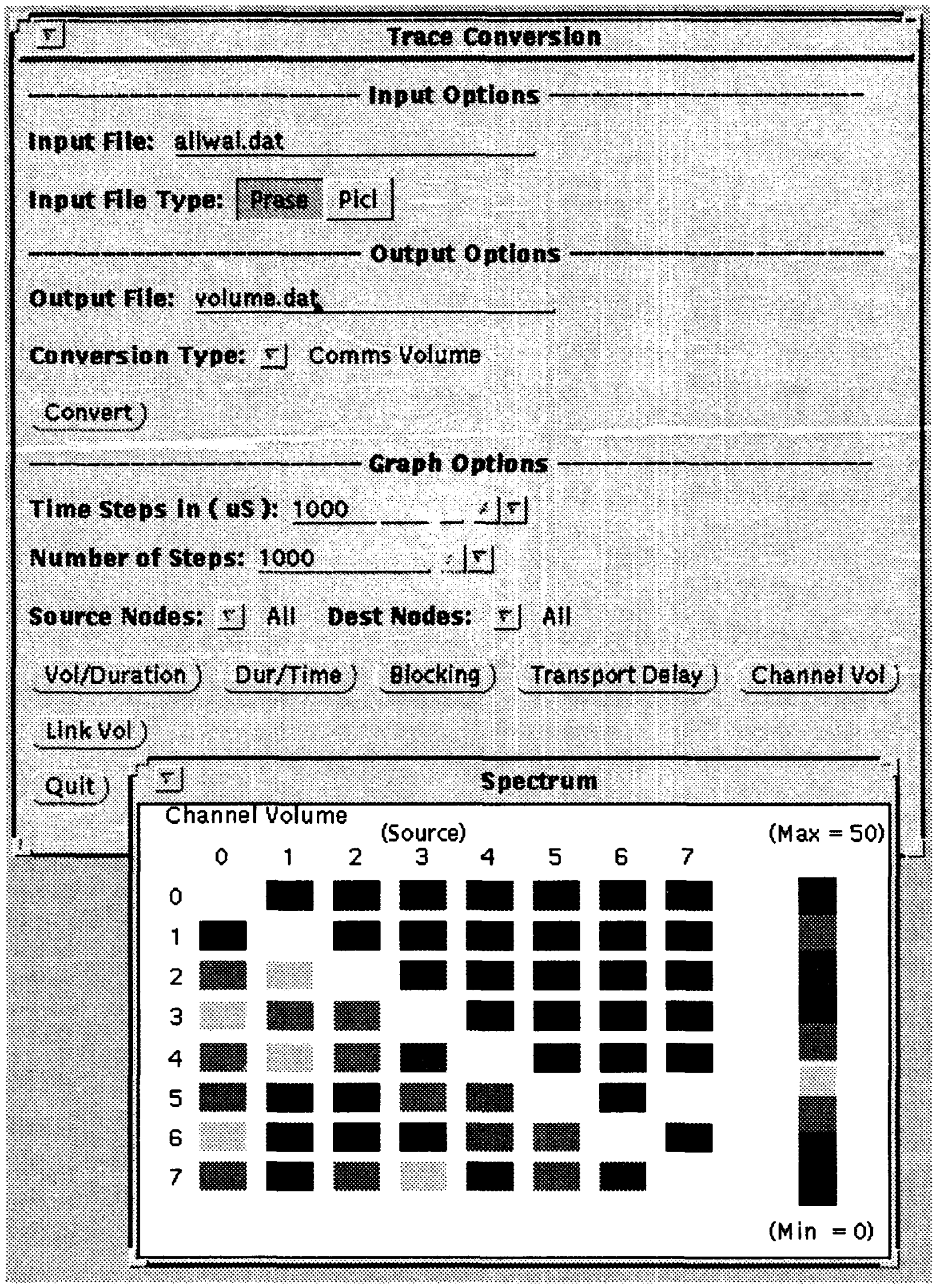
(82). Since gnuplot is freely available on work stations at AFIT and the plotting requirements are limited, gnuplot was used as the plotting package. However, users are free to select their own if they desire. The file output of the conversion program can be readily used by Gnuplot(82) to produce hard copies of analysis data. This implementation allows displays from multiple traces to be depicted at the same time.

6.4 Overview of Analysis Tools

The central control panel is shown in Figure 27 and allows the input of a trace file name and selection of conversions and display type. The input file can be either in the standard PRASE or the PICL trace format. In the current implementation, the PICL trace conversion is limited to only the communication primitives. The PRASE file input is compatible with all event types and can use additional Task/Phase events to more accurately determine processor idle time.

The analysis tool provides a number of output files that can be generated from input trace files and are listed below. The output from these conversions are in ASCII text format and are suitable for input into graphical display programs for alternative analysis.

- *Complete Trace to text:* This selection allows a complete PRASE trace file, recorded in binary format, to be converted to text. This is useful in determining timing and event types contained in trace files.
- *Selected Send/Recv's Only:* This selection allows the conversion to text of communication events contained in a PRASE trace file. The user can specify which node pairs are of interest or include messages between any processor pair.
- *Selected Communications Time Distribution:* Used to produce a data file showing the relative distribution of communication times.


Trace Conversion

Input Options

Input File: allwal.dat

Input File Type: [✓ Prase] Picl

Output Options

Output File: volume.dat

Conversion Type: Comms Volume

Graph Options

Time Steps in (uS): 1000

Number of Steps: 1000

Source Nodes: All Dest Nodes: All

Figure 27. Control Panel for Analysis Tool

- *Communications Time versus Execution Time:* This display shows trends in the communication time relative to the program's execution. This information can be used to detect periods of execution where the message congestion increases average message delay times.
- *Routing Information:* This selection allows the interesting event markers that are inserted into mission routing (14) trace files to be converted to text format. This produces a file containing all co-ordinate tuples visited during the search process. It can be used as input into any suitable graphics package such as MATLAB (41), or the Explorer (75) graphical display environment.

The system also includes a number of graphical displays that are selected from the control panel, including a interface with gnuplot (82).

- *Transportation Delay.*
- *Communications Blocking Display.*
- *Communication Volume Display (Logical and Physical).*
- *Concurrency Profile Display.*

The following sections outline the displays introduced by the analysis tool.

6.4.1 Transportation Delay . This display is designed to focus attention on periods where communication times were particularly excessive. This can be a result of long blocking times due to poor correlation in time, message size or channel conflict. The message transportation times are shown relative to the execution time of the program in this display.

The display can be adjusted to show all communication events or any combination of nodes and transportation times. For example, it is possible to show only communication events that were received by nodes that had been blocking for a period greater than the average transport delay. This highlights periods where dependencies in the program under examination caused low utilization.

6.4.2 Communication Blocking Display. The communication blocking display depicts the distribution of blocking time experienced during execution. The display records the time node spent blocking with the *csend* and *crecv* communication primitives. This display shows how well synchronized in time the inter node communication events are with respect to each other.

Many applications do not make use of blocking type communication primitives, for example Spectrum (5). In these cases, combinations of the *isend*, *iprobe*, *crecv* and *irecv* communication primitives are used. For this animation the *irecv* does not present a problem since the processor is continuing with useful work and is effectively not blocked. However, the common usage of the *iprobe* to detect message arrival and then execution of the *crecv* command to read the message presents a problem. The *iprobe* command in this case is effectively used as part of a busy waiting loop. Thus the processor is actually blocking for communications.

To overcome this problem of artificially high utilization, the display provides an optional selection which includes input from Task/Phase trace events. By placing task event markers around the section of code that executes the busy loop, we can ensure that the blocking display correctly reflects the system activity. In this mode, blocking is counted during periods of busy waiting.

6.4.3 Communication Volume Display (Logical interconnections). The logical communication volume display depicts the number of messages sent between processors during the execution of the program. The display is formatted as a matrix (see Figure 28) with all sending and receiving nodes listed. In this way the volume in a particular direction can be determined. The display uses both a pie chart representation of the message volume and color coding to present the information. This particular display uses data collected for a Spectrum VHDL simulation (see Chapter V).

6.4.4 Communication Volume (Physical Interconnections). This display is identical to the logical communication volume display in format. This display however, includes message volumes only on physical processor interconnections. Most current parallel processing systems utilize a

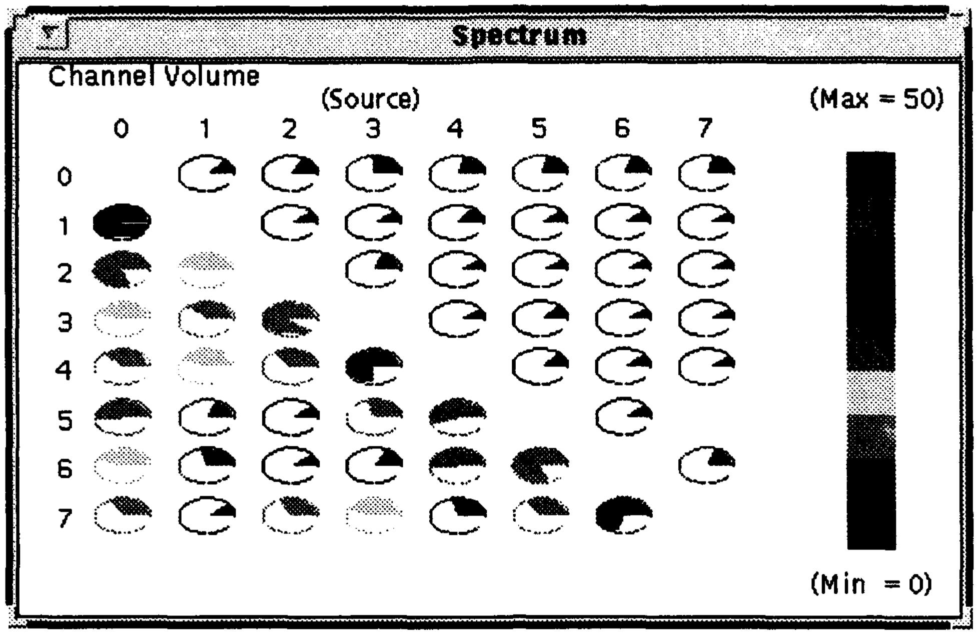


Figure 28. Communications Volume Display

subset of the links that would be implemented in a fully connected system due to cost constraints. The result is that communications between processors must share communication channels. In the current AAARF implementation on the iPSC/i860, a hypercube is used for interconnections. However, others exist such as tree, mesh, cube, prism, and ring structures. This animation uses a physical processor interconnection specification and routing map to determine message volume on a particular physical link. The current implementation includes only the definition for the hypercube architecture and could be later expended to other topologies.

6.4.5 Concurrency Profile Display. One feature that is present in many other systems is a display that provides a summary of the total processor utilization. This information is often depicted as a bar graph showing the number of processors utilized over the entire program's execution. It is particularly useful in determining what percentage of the program executes sequentially and what level of concurrency is achieved in the parallel sections. From experiments with various algorithm classes it became apparent that this form of display was lacking in AAARF. After experimenting with a number of different displays it was found that the simple bar graph display used by PICL provided the clearest presentation.

6.5 Trace Data Conversion

To allow greater flexibility in data collection for the AAARF system, a conversion program was developed. A conversion tool has been combined into the AAARF menu system and produces PICL comparable execution traces from data collected using the PRASE (42) instrumentation system. The PICL system is currently the most widely used instrumentation suite and is compatible with the complete range of Intel parallel systems. Further work on this conversion program was not undertaken as a result of product releases scheduled by Intel Corporation in the last quarter of 1993. The latest set of programming tools available for the Intel Paragon, ParAide (39) include trace data collection. The format used is that developed as part of the Pablo (62) project. This tracing capability is built into the Paragon operating system and thus is likely to make other trace formats redundant on the Intel series of parallel systems.

The conversion process, as implemented, is restricted to the communication and utilization event types, due to limited support for high level algorithm event tracing incorporated into PICL. The PICL system differs from PRASE in that it does not redefine the standard Intel communication calls to include event tracing but requires the user to use specific PICL calls. In addition, a number of initialization calls must be made.

6.6 Summary

The static analysis displays examined in this section including transportation delay, communication blocking display, communication volume display (logical/physical), and concurrency profile add significant capabilities to the AAARF system. Without this type of display, detailed analysis of program execution cannot be performed easily. The basic system developed for this experimentation provides adequate capabilities but does not allow flexibility in the display format. Environments such as Pablo (62) provide a user friendly design system to produce custom display formats. This approach would appear a flexible alternative to allow users to build onto the basic displays included in the AAARF analysis tool. Implementation of such an animation environment is discussed in Chapter IX.

Building flexible display formats is only half of the problem. To ensure that the resultant animations reflect actual system behavior and the data collection requirements can be met, we must examine the instrumentation system. The accuracy of the data presented by the analysis displays and the other AAARF animations is dependent on accurate trace data. The next chapter examines the impact of event tracing and a number of related restrictions.

VII. Performance Analysis of Trace Data Collection System

7.1 Introduction

This chapter examines the problems inherent in trace data collection systems. Analysis of the tracing overhead ensures that the level of observation, allocated resources, and perturbation effects are appropriate for the planned visualizations. Specifically this analysis aims to quantify the tracing overhead introduced by PRASE (16), the data collection software used by AAARF. Instrumentation and tracing of executing algorithms can consume significant resources and impose performance variations that can render the collected data irrelevant for detailed examination of performance. The secondary goal is to determine the scalability of PRASE and other data collection systems and in addition real time data collection capability. This type of analysis indicates the practical issues involved in pursuing algorithm animation for current massively parallel systems.

7.2 Instrumentation Overhead for PRASE

The data collection system introduces an additional overhead into the execution of the program under examination. This is a result of the extra code inserted into the target program to record interesting events (16). The significance of this overhead is dependent on particular aspects of the program under examination and the characteristics of the underlying algorithm. Both these characteristics effect the volume, type, and frequency of event data collected. The overhead introduced by AAARF can be considered to include two basic components.

- The time required to store time stamped events in local memory (Computation Overhead).
- The effects of additional message traffic required to dump trace data to host (Communications Overhead).

The following sections examine this overhead in detail. The analysis is not only applicable to PRASE but to all software based data collection systems. Only with the use of non intrusive hardware based data collection can these effects be ignored. Hardware based instrumentation systems are discussed in detail in (50). The implementation used in PRASE contains the same fundamental steps used in both PICL, and Pablo (62).

7.3 Computational Tracing Overhead

It would initially appear that the computational overhead would be quite small, however, the actions required to record an event are relatively complex. The minimum data required to specify an event includes the following items:

- Recording node.
- Start time of event.
- Time of event completion.
- Event type or format information.
- Event data fields.

To minimize trace volume, most data collection systems, including PRASE, use events with duration. For example, events such as *send* and *recv* require an operating system call. Depending on the availability of message channels for sends, and appropriate incoming messages for receives, these calls can block execution of the node program. Since the duration of this blocking is important to our analysis, it must be accounted for by trace events. If only a single time stamp was used for each event the tracing system would be required to create two event records, one for start and one for complete, doubling the trace data volume. The secondary advantage of this approach is the ability to simply redefine communication primitives to an alternative function that includes the event generation code.

To actually record the event we must execute instrumentation code that generates this information and appends it to the current list of event records. Since we are only examining the computational overhead in this situation, we can assume that all events are to be stored in local memory. The most costly aspect of the event tracing is determining the actual time stamps for the events. Obtaining the current local time requires operating system calls on the iPSC/2,iPSC/860, and the Paragon. On the iPSC machines, the node clocks are not synchronized and must be adjusted to reflect the correct global time. This adjustment is necessary to ensure that events on different nodes are referred to the same time frame. Thus, the minimum functions required to record an event are:

- Procedure call overhead.
- Two calls to system clock.
- Two long integer additions.
- Two integer memory to memory copies.
- Memory copy for event data (1 to 1K bytes of data).

Both the node operating system and memory cache contents greatly effect the actual duration of this event recording. It is therefore best to determine the actual overhead imposed by experimentation. To analyze the effects and size of this overhead, the event times from a number of application programs were examined. By recursively storing two thousand events with only the loop overhead, then subtracting the cost of this overhead, it was possible to determine the actual event recording duration. This test was conducted on both the iPSC/2 and the iPSC/860 and the results are contained in Table 7.3.

System	*IE*(type,data1,data2)	Comms Event
IPSC/2	500 μ S	280 μ S
IPSC/860	300 μ S	170 μ S

Table 1. Average Time to Record an Event (Experimental)

This is not the only variability introduced into the event trace data. The time function calls on the iPSC/2 and iPSC/860 have been shown to only be accurate to within a $50\mu S$ time window

7.7.6 Results Summary. The figures produced above provide a graphic illustration of

(40). When this is combined with the effects caused by the tracing routines, significant distortion of event records can occur. As would be expected, this variation can cause events to be recorded out of order, a common problem with all current software based instrumentation systems. This has begun to be addressed by post processing techniques (71). AAARF does not currently account for this overhead or out of order events caused by distortion on event times.

7.4 Communications Overhead

The real-time data collection facility in the iPSC/2 and iPSC/860 implementations of AAARF introduces the additional overhead of transmitting events to the host. This overhead includes the cost of sending an event message, or messages to the host using iPSC message passing primitives. The greatest disadvantage here is the fact that synchronous message passing is required between the nodes and the host. Thus the node processor must wait for connection to the host and the communication to complete. Attempts to use asynchronous message passing to the host have proven unreliable in my experiments with PRASE. The limited message buffer space on the node combined with frequent blocking to the host, causes the available message buffers to be depleted. Based on the current implementation of AAARF, the following event transfer times were determined experimentally.

- Average time to transmit event to Host for iPSC/2 = 9.5 mS.
- Average time to transmit event to Host for iPSC/860 = 6.4 mS.
- The important parameter here is that there is no upper bound. If the Host is swamped with event packages from the nodes any particular node can wait indefinitely.

As can be seen from the above data, events sequencing information is nearly completely lost during event dumping periods. The additional effect of this overhead is to reduce the effective bandwidth of the communication channels. However, since little useful information can be gained during this period and the program has already been significantly perturbed, this additional effect is less critical. This is a particularly important result since AAARF does not provide data buffering of all event types. This implementation decision made by Williams (84), results from the strongly typed data buffer implemented in PRASE. As a result, any *IEdata()* events are sent directly to the host and if this particular event type is used, this timing error is introduced into the trace result for every IEdata() call. Implementations should utilize separate *IE()* calls to allow accurate time tracing when required. IEdata calls should only be utilized when storage of large arrays are required and event timings are not the focus of analysis.

7.5 Combining Effects of Overhead Types

PRASE allows the two types of overhead to be traded against one another using variable trace buffer sizes. With a trace buffer sized to one, each event is sent directly to the host. This avoids the problem of limited node memory for trace files, a very constrained resource. However, this introduces significant variations in the program's execution, as seen previously. The main problem with this technique is that all data is collected on the host and the trace data must be transferred to the host across the same communication channels used by the program under examination. The actual time period for this activity is mentioned in the previous section. To better understand the effects of this problem on complete application programs, a number of sample progr. ms were executed with no event buffers. The effects of this can be clearly seen in the increase in execution time shown in Figure 29. This particular figure provides data for the VHDL simulation program with different levels of event tracing. While trace information generated in this manner is unsuitable for analysis of performance, this level of tracing remains a useful feature for debugging.

To avoid these significant perturbations, trace buffers can be used. The size of these buffers are limited only by available node memory. This form of tracing eliminates the communications overhead and only the much smaller computation overhead remains.

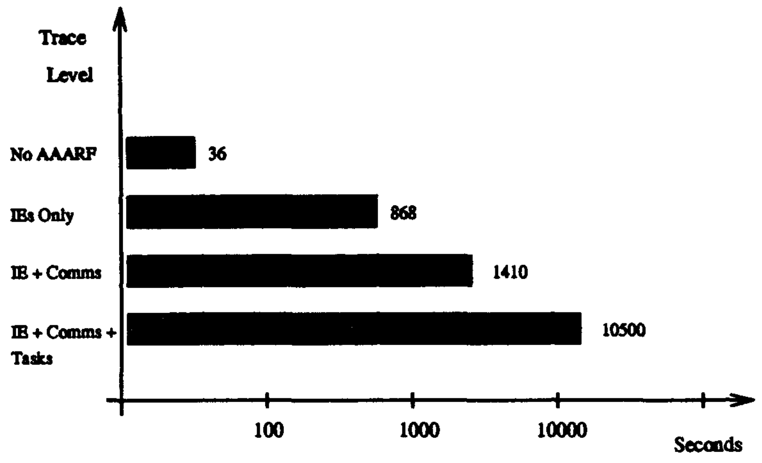


Figure 29. Execution Time with No Trace Buffer

While the use of buffering limits these perturbations in the resulting trace, it introduces problems with trace volume. This trace volume problem has been a focus of much research activity (51). The key to reducing this volume is to produce more compact trace formats and allow intelligent selection of events to record. This can be further enhanced by combining individual events into higher level composite events.

Figure 30 shows the volume of trace data collected for a sample range of application programs. The information shown in this figure has been normalized based on the program's total execution time.

This event volume can be divided into that required for the basic communications event tracing and that required to produce the algorithm animations. The volume of trace data produced is thus dependent on the type of algorithm under examination and the level of tracing utilized. Figure 30 also shows this distribution for the VDHL simulation (5) which tends to represent a communication bound program.

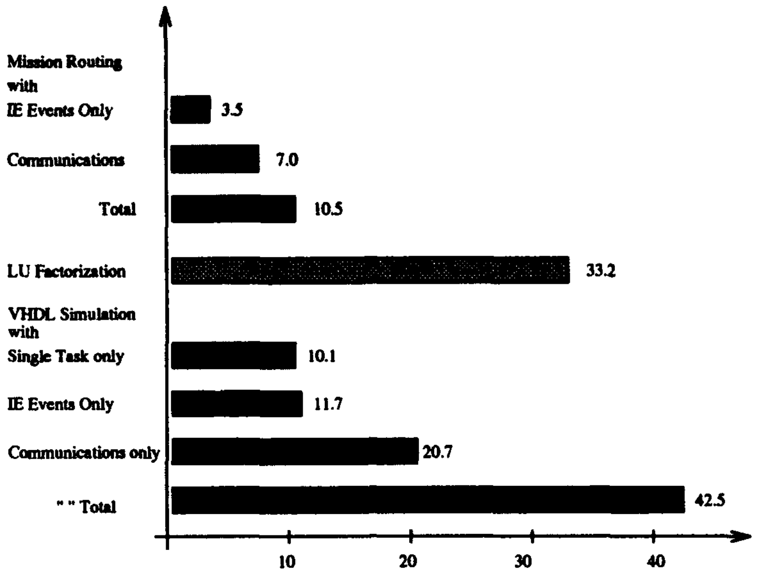


Figure 30. Normalized Trace Data Volume (KBytes per Node per Second)

7.6 Trace Data Volume

Parallel processing systems are rapidly increasing in both the number of processors they contain and in the execution speed of their processors. The volume of trace data collected during the execution of an application on these systems becomes a considerable problem. An estimate made by (68) predicts that the one minute execution of SOLOMAN (68:544) benchmark on a 1024 node i860 will generate trace files in excess of 500Meg bytes. Recent research efforts have attempted to address this problem, however, no significant progress has been made. As a result most tracing systems are limited to tracing only segments of the program's execution. This is the typical approach taken by PICL (23).

PICL is limited to tracing data up to the maximum volume the processor's node memory can store. Once this limit has been reached the tracing is discontinued. This limitation is compensated for by the system allowing the trace to be enabled and disabled at different points in the program's execution. An alternative approach was incorporated in the Ames labs visualization system (57) AIMS, discussed in Chapter 2.4.2. The AIMS system allows the users to specify what data is to be collected, by allowing the insertion of a number of predicate statements. These predicates which are evaluated as trace data is collected, only record events that meet the user's specified conditions.

The AAARF system approaches this problem by allowing the processor nodes to dump trace data during program execution, to the host processor. While useful from a conceptual understanding or debugging perspective, the perturbations introduced limit its usefulness in detailed analysis. It is important to note that AAARF, like PICL, is already c[illegible]f producing relatively accurate traces when they can be contained in the node memory. The following section discusses additions to the AAARF system, that have been included to assist with reducing trace data volume.

7.6.1 Trace Data Minimization. The PRASE system uses a combination of C language *#ifdef* statements and *include* files to redefine the Intel system calls for event tracing. This enables programs to be instrumented with the minimum of effort, unlike PICL which requires the Intel system calls be replaced with calls to the PICL library functions. The inclusion of trace data collection into the operating system for the Intel Paragon should significantly improve the ease of data collection. However, the implementation used for the Paragon system does not address trace volume minimization. In addition, tracing on the Paragon is currently limited to system level events only (39).

7.6.1.1 Limiting Event Tracing. The level of tracing provided by PRASE can be divided into system level function calls and interesting algorithm events. Tracing of both these events during the program's execution produces the largest possible trace file for any given program instrumentation. The interesting events cause a particular problem as they can contain entire algorithm data structures such as snapshots of an array's contents. To alleviate this problem, it was decided to divide the trace types into logical blocks that could be enabled or disabled by the user. A number of alternative methods for reducing trace volume, including trace predicates and trace file format changes were examined. Dividing trace events into logical classes was considered the most effective method due to the fixed data requirements of most animation displays. For a particular animation, a set of events will be required to correctly render the desired image. For example, utilization displays are only meaningful if all periods of inactivity are recorded. There is a separation however, between the system and algorithm events since they update different animation displays. In particular an application specific display may only require a single interesting event marker to produce a correct animation.

The following technique has been developed to allow the level of event tracing to be varied with the minimum of user effort. Every application program that utilizes the PRASE data collection system uses an *#ifdef PRASE* statement to allow inclusion of the *phase.h* file. This file contains the *definition* statements that redefine the Intel system calls to the instrumented PRASE calls. For example, *csend()* becomes *prasecsend()* . In addition to the basic PRASE compile flags, additional definitions have been included that allow control of the data collection. These modifications to the prase.h file allow user selection of the following event tracing levels.

- *Particular message passing.* Use compile flag **PRASE** and replace interesting communication pairs with *prasecsend* and *prasecrecv.* This is also possible for non blocking communications.
- *All system level events.* Use compile flag **PRASE and ALL.** This provides tracing of all communication events and Intel specific system calls.
- *High level algorithm events.* Use compile flag **PRASE and ALG**. This provides event tracing of only user defined algorithm events.
- *Tracing all events.* Use compile flag **PRASE, ALL, and ALG** to enable tracing of all possible event types defined for the application program.
- *Collection frequency.* Some application programs, mission routing for example, allow the frequency of the trace collection to be defined. In this case a **tell_count** variable is included to allow specification of the data collection rate. This variable only effects the *IE* and *IEdata* event types.

These techniques require further refinement and a more convenient user interface. However, they do provide the ability to reduce the trace volume by orders of magnitude in many cases. For example, with the mission routing problem (14) the search space diagrams are updated after every 200th node expansion and provide that order of reduction in the trace file size. If significant interest is shown in particular sections of the execution, shorter tracing periods with more detailed data collection can then be focused on areas of interest. The insertion of further conditional statements within this framework is also possible.

7.7 Future Impact of Data Collection Limitations

Research into the development of new and interesting animations provides a virtually unlimited scope of possibilities. Many of the animation systems are beginning to address the problems of display scalability for massively parallel processing systems (62). However, the ultimate acceptance and use of these displays is dependent on the ability to collect the associated trace data. From the previous discussion and other research examined, it would appear that while reductions are possible, a significant limitation remains in our ability to provide detailed views of long execution periods. This section examines the question of whether system level event tracing is applicable to massively parallel systems.

7.7.1 System Focus. To examine this question, an analysis of parallel computing systems and their capabilities is required. While there are many systems currently available on the parallel processing market, there is increasing convergence on the MIMD architecture. The Connections Machines CM-5 (81), the just released Cray Research's T3D, Intel's Paragon, and others are increasingly utilizing commercially reduced instruction set CPUs in their parallel systems. Processors such as the DEC Alpha (11), Intel i860XP (37), and Sun Sparc processors (31:E-6) are by far the most commonly used processors for these systems. The relatively high performance of these devices coupled with interconnection bandwidths of between 2 and 400 MHz and communication latencies of $100\mu S$, lead to a range of very course grain systems. This course grain structure simply implies that substantial processor elements are located at each node. As a result, the algorithms implemented on these systems are aimed at utilizing this capacity by decomposing the problem into larger individual chunks. In a fine grain approach, the problem is decomposed into smaller less complex sections and can extend even to the bit level.

The recently released Intel Paragon (39) system represents this emerging broad class of MIMD machines with high speed reduced instruction set (RISC) processors and a mesh interconnection network. Lower order interconnection systems such as meshes are being utilized to limit the number of physical interconnections as the number of processing elements they contain increases. The Paragon is used as the baseline system for these calculations as it provides an approximate model for many present day systems. There are currently over 50 Paragon systems installed, with single installations containing up to 2800 processing nodes per system. The Paragon system recently acquired by Wright Patterson Air Force Base contains 240 processing nodes. This particular machine is the likely target for future AFIT parallel algorithm research. This represents a significant increase over the performance of the current AFIT system.

To provide an estimation of likely trace data volumes on the Paragon system, we must consider three basic components: system performance characteristics; event storage; and algorithm behavior.

7.7.2 Processing System Performance. The parallel processing capability provided by the Paragon can be represented by the following simplified set of operating parameters

- 50 MFLOP processing nodes.
- 50 μ Second transportation latency.
- 200 Mhz internode communication bandwidth.
- Node operation system time slice $\frac{1}{10}$ of a second.

These characteristics provide a representative picture of a large parallel processing system in current use. Note: A linear model approximation of communication time is used. This approximation is frequently used for determining decomposition strategies and estimating program performance and is discussed in detail in (45). It is interesting to note that the number of hops required to reach message destination no longer plays a significant role in determining performance. Only a 5% variation is experienced in communication time between any two nodes in the system. This does not however, account for message conflict for a particular channel.

While considerably dated by today's standard, AFIT also utilizes an Intel iPSC/2 system. To provide a comparison, the parameters for this system are:

- 0.5 MFLOP processing nodes.
- 900 μ Second transportation latency.
- 2.8 Mhz internode communication bandwidth.

7.7.3 Event Storage Requirements. To model the data collection and storage requirements, the following representative event sizes from PRASE are used. These values are comparable with that achieved when using the PICL (28) data collection system in compact format. For this examination it is considered that only the following listed events are to be recorded during execution. This provides the minimum subset required to produce basic system level displays. These events are used to generate Feynman, Utilization, Kaviat, and other similar animations.

- Communication primitives (25 bytes per event). - V_{comms}
- Operating system calls (10 bytes per event). - $V_{context}$
- Message buffer state (10 bytes per event). - V_{queue}

The communication primitives require trace information to be recorded by both the source and destination processing nodes. The Intel Paragon includes a complex Unix operating system on each processor node. The processing nodes are not solely allocated to a single user and can belong to more than one user partition during execution. As a result, the overhead imposed by the operating system and any task owned by another user must also be included in animation displays. The iPSC/2 uses the simpler N operating system on each node processor and only one user can be allocated a processing node. While it is possible for iPSC/2 nodes execute more than one task, these tasks belong to the same user application and thus this additional information is not vital.

7.7.4 Approximate Algorithm Model. The programs provide a more difficult element to model since their granularity significantly effects the number of events produced during execution. For communication intensive tasks, message passing can require over 80% of execution time. In applications with higher degrees of parallelism, communications can account for less than 5% of execution time. This is further complicated by the fact that both the communication patterns and message size determine the actual number of communication events that occur per unit of communication time. From analysis of the applications examined as part of this research, 10% of the execution time would be a conservative estimate of the communication time for a representative parallel application. If we assume that during this period, 50% of processors participate in communications with messages of approximately 1K byte, it is possible to approximate the communications load.

- Program message passing overhead estimated at 5%, 10% and 20% of program execution time.
- Average message length 1K Byte.
- Message passing pattern with 25% of processors pass messages to one other processor during communication events. This is referred to as the average message passing participation rate.

7.7.5 Trace Volume Calculation. This section calculates the trace volume for a number of program scenarios. To determine this information, the following parameters and equations are used in the calculation of trace volume:

- Total tracing duration in seconds - T_{trace}
- Average message length in K bytes - Av_{mess_length}
- Number of processing nodes - P_{nodes}
- Internode communications latency -T_{comm_delay}
- Interconnection link bandwidth in MHz - BW_{mhz}
- Operating system time slice in seconds - OS_{sw}
- Percentage of execution time for message passing - $M_{overhead}$
- Message passing participation rate as a percentage of total system nodes - $M_{participation}$
- Trace volume produced by communications event in bytes - V_{comms}
- Trace volume produced by operating system context switches in bytes - $V_{context}$
- Trace volume for message queue size statistics in bytes - V_{queue}

Average Message Duration $AV_{mess} = .$

$$T_{comm\_delay} + Av_{mess\_length} \times \frac{1}{BW_{mhz}} \; Seconds$$

Total Trace Volume $Total_{volume} = .$

$$\left(\frac{M_{overhead}}{100 \times AV_{mess}} \times P_{nodes} \times \frac{M_{participation}}{100} \times 2 \times (V_{comms} + V_{queue}) + V_{context} \times P_{nodes} \times \frac{1}{OS_{sw}}\right) \times T_{trace}$$

$M_{overhead}$	Number of Processors						
	8	32	128	256	512	1024	4096
0.05	128K	512K	2.0M	4.0M	8.2M	16.4M	66M
0.10	255K	1.0M	4.1M	8.2M	16.4M	33.8M	132M
0.20	509K	2.0M	8.2M	16.3M	33.8M	65M	260M

Table 2. Trace volume for one second of execution on Intel Paragon.

Using this equation the trace volume for a number of scenarios can be determined. The scenarios examined include tracing on the Paragon for 1 second, Paragon for 1 hour, and iPSC2 for 1 second.

Paragon One Second Scenarios. This example shows the volume of trace data generated by an Intel Paragon in one second of operation. On a 256 node system this would be enough time to multiply two 1000 × 1000 matrices of floating point numbers. Table 2 shows the results of the trace volume calculations.

T_{trace} = 1 Second $\quad$ Av_{mess_length} = 1K bytes

T_{comm_delay} = 50μ seconds. $\quad$ BW_{mhz} = 200Mhz.

OS_{sw} = 0.1 Seconds.

Paragon One Hour Scenario. When expanded to more realistic sized problems, the data storage requirements become immense. Table 3 shows the full extent of the problem. For this calculation, T_{trace} = 3600 seconds, and all other parameters are as previously specified. The table values represent the number of bytes of storage required. It is obvious from the values in this table that the program tracing in this manner would be virtually impossible. However, this raises the question of how the user detects errors in program behavior. If your program simply *changes* the system 20 minutes into execution, what data can be collected at that point to assist in debugging. This is particularly relevant if we are trying to examine the execution of the algorithm, not simply its mapping to a particular system. In this application, we are looking for logical errors in design, not bugs in implementations. Unless we are allowed to examine these longer periods of execution we are unable to determine such features as data convergence or other progress properties.

$M_{overhead}$	Number of Processors			
	128	512	1024	4096
0.05	7.4G	29.5G	59G	240G
0.20	29G	118G	230G	940G

Table 3. Trace volume for one hour of execution on Intel Paragon

iPSC2 One Second Scenarios. The reason why trace volume is now becoming more of a problem can be seen from the result shown in Table 4 for the iPSC/2. Many of the animation systems in use today were initially developed for a system such as the iPSC/2. Their poor internode communication performance and slower execution rate lead to a lower rate of event generation. A good example in the change in performance can be highlighted by floating point performance of newer systems. For floating point multiply operations, the i860XP processor in the Paragon is 40 times faster than the iPSC/2. This was measured using the LINPACK benchmark (13).

T_{trace} = 1 Second Av_{mess_length} = 1K bytes

T_{comm_delay} = 900μ seconds. BW_{mhz} = 2.8Mhz.

OS_{sw} = 0 Seconds.

$M_{overhead}$	Number of Processors		
	1	4	8
0.05	3.5K	14K	56K
0.20	14K	56K	224K

Table 4. Trace Volume for One Second of Execution on Intel iPSC/2

These lower event generation rates are, to some extent, compensated for by an increase in total execution time. However, the problems examined using these systems were in most cases smaller in dimension and thus often required only similar total execution times to that of more current applications.

7.7.6 Results Summary. The figures produced above provide a graphic illustration of the scope of the trace data volume problem. The values calculated correspond well, in terms of total volume, with other similar results (68). When we consider that execution of many parallel applications requires tens to hundreds of system hours, this level of tracing becomes impractical. From these results it is possible to speculate that this form of event tracing will be relegated to examining the execution of small program kernels, not complete applications. While this is not a completely new concept, it highlights the futility of producing highly abstract scalable displays for this type of event tracing. If this level of tracing is applied to a larger system it will be to examine very detailed behavior for short periods, on a subset of nodes. The most obvious application here would be debugging where the user is likely to require examination of each state change. Thus, high level displays must be generated for high level event data, not produced from abstracting detail from low level event tracing. A top down approach to event collection and display, rather than the more prevalent bottom up approach should be adopted.

It is important to focus animation techniques on increasing the abstraction of lower level algorithm features. This would not only provide a reduction in trace volume, but allow more direct correspondence of events to algorithm behavioral components. These events are also simpler to compare with program constraints to monitor correct operation of the program under observation. The occurrence of simple events such as messages and context changes cannot easily be related to program progress, particularly if the execution is non-deterministic or recursive in nature.

7.8 Real Time Animation Requirements

If we consider the additional requirements of including real time data display, the problem of event tracing becomes more difficult. In this case, the parallel processing system's input/output capabilities must be included in the calculations. To provide this capability, the trace data collected at each node must be gathered and passed to an external display system. For the current iPSC2 and

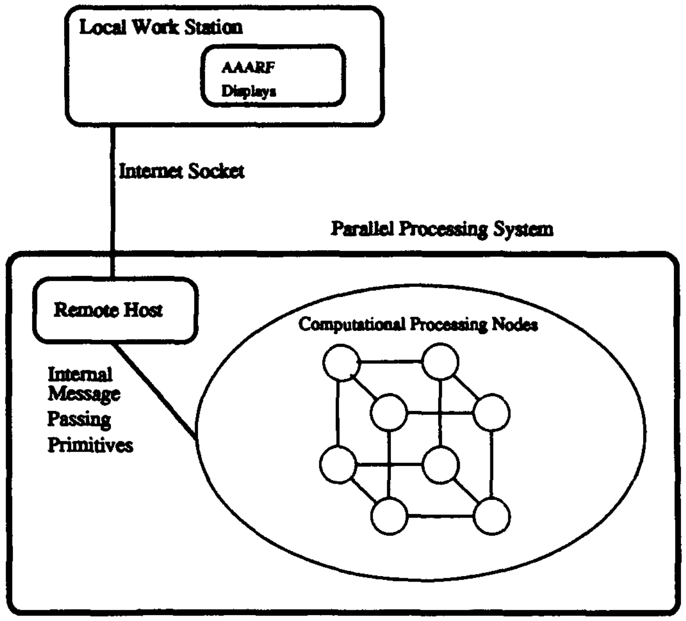

Local Work Station
AAARF Displays
Internet Socket
Parallel Processing System
Remote Host
Computational Processing Nodes
Internal Message Passing Primitives

Figure 31. Interconnections for Data I/O on Intel iPSC2/860 System

iPSC860 this involves sending the data directly to the host processor using the Intel communication primitives. This is then passed to the display terminal using Unix socket communications. This relatively simple model, which is implemented in AAARF has the obvious bottle neck at the host processor. Figure 31 depicts this configuration.

If we expand this to the more advanced Paragon system, the Host node is no longer included. In this system the processing nodes are arranged into sets of **service** and **computational** nodes. Processing nodes in the service partition are used for interactive users and computational nodes allocated to user programs. Any processor in the system can be allocated an I/O interface directly by the addition of a network interfacing card. However, most systems utilize I/O through separate processing nodes allocated to an I/O partition. Figure 32 illustrates a typical Paragon system. Note that the number of nodes in the vertical direction are limited to 16 nodes.

A standard I/O system for a 512 node Paragon would consist of one Ethernet connection and one HiPPI port (53). For the 2800 node system at Sandia Laboratories, two HiPPI interfaces are used. Thus while the demand for high I/O data rates exist, cost in most cases limits the actual

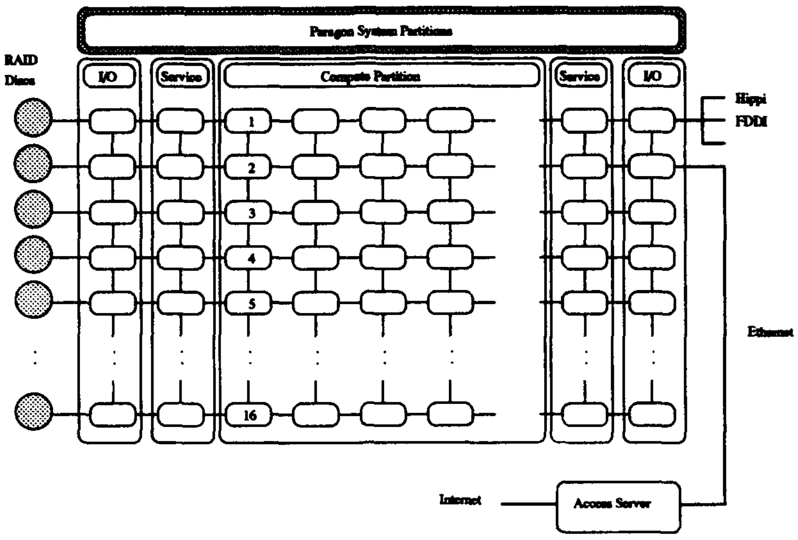


Figure 32. Paragon System Configuration

total I/O bandwidth of the system. The maximum data effective transfer rates measured (53) for these interconnections are:

- HiPPI Interface - 66 MByte per second.
- Ethernet - 2 MByte per second.

To actually localize trace data to nodes that have an interface connection, the trace information must be passed via the processor communication mesh.

7.9 A Manageable Real Time Event Rate

The overhead produced by the message traffic required for real time execution tracing provides a significant limitation on this type of tracing. The results shown in Figure 29 indicate that even for a small scale system, the effects of this type of tracing are enormous, in many cases causing orders of magnitude increases in execution time. As the dimensionality of systems increases, the problem is compounded. This section estimates what level of tracing is possible on larger systems such as the Intel Paragon. For the processing model, we use the 240 Node Paragon located at Wright Patterson AFB, since this is one of the targets of our research at AFIT.

This Paragon system includes the following data transfer capabilities:

- Two Ethernet connection.
- Five HiPPI Interface.
- One FDDI network connection.

Currently AFIT is connected to the Paragon via Ethernet only and thus real time animation will be very restricted but still possible in limited cases. Animation of algorithms for execution monitoring and debugging is likely to remain on Ethernet for the near future and is particularly relevant, since remote hosting on other distant systems will remain a requirement. The higher speed interfaces will be utilized more heavily by applications producing graphical results for direct visualization of data. However, with later expansion of the network, or by utilizing co-located workstations, the FDDI or HiPPI interconnections could be used. Considering a representative application on the Paragon, it would be likely to have access to the following system resources:

- A compute partition containing 100 to 200 processing nodes.
- The user logged onto the system via Ethernet to a node in the service partition.
- Program data stored on RAID disks connected directly to nodes in the I/O partition.
- Access to a HiPPI or FDDI node in the I/O partition.

The computational system model is thus a mesh with service and I/O nodes allocated along one or both vertical edges of the mesh. The Paragon limits the vertical dimension of the mesh due to physical requirements. Figure 32 shows a block diagram of a typical Paragon system.

7.9.1 Data Transfer Rate. The actual transfer rates of the interconnection system between the graphical workstation and the target system significantly limits our real time animation capabilities. It is pointless to have real time animations that lag the execution by such an extent that simple *ftping* of stored data files could produce the same result.

Determination of the data transfer rates that can be supported by the systems I/O devices provides an upper bound on trace data requirements. This in turn, provides an indication of the appropriate level of abstraction for the animation. This type of animation is not restricted to simply examining program execution and can include visualization of partial solutions.

7.9.2 For Ethernet Only Connection. For this calculation we will assume that our program has exclusive access to a particular service node and I/O node with a single Ethernet port. The Ethernet system has a nominal bandwidth of 10 MB/sec, however, the practical upper limit on transfer rate is much lower. The actual transfer rate is effected by traffic load and message sizes used. For these calculations a bandwidth figure of 1.2 MBytes per second is used, this represents the data rate that could be obtained when the network is lightly load. This value was determined from experimentation with the current AAARF system and assumes a combination of individual events. It is important to remember that this can range greatly depending on data packet size. Effective bandwidths of less than 10K Bytes per second have been measured.

Based on these bandwidth values, the maximum trace volume generated per second by the system would be limited to 37.5K bytes, assuming 32 byte events. Dividing this by the number of processing elements in the system the corresponding figure per processing node can be obtained.

- Assuming 240 processing nodes.

$$Trace\ Volume\ per\ second\ per\ node = \frac{37500}{240} = 156\ events.$$

From previous analysis shown in Figure 2, approximately $\frac{16K}{32} \times 240 = 120K$ events per second could be generated by such a system. Thus, it could be expected that under these conditions we can achieve only approximately one thousandth of the resolution required when only displaying communication events. This factor is based on 240 processors and scales directly with the number of processors. Therefore, the resolution must be reduced by the same factor as the number of processors are increased.

At this data rate, the cost of collecting the trace data at the I/O node is minimal. The 200 MB interconnection between processors results in less than 1.0% loading on the channels adjacent to the I/O node. This would however, require intelligent re-combination of events as they are sent to the data collection node. If each event was transmitted separately with no buffering, the message latency would play a greater role, leading to an unacceptable $37500 \times 50\mu \times 100 = 187\%$ utilization of the inter-connecting channel to the Ethernet I/O node.

7.9.3 For HiPPI or FDDI Network Connections. The FDDI and HiPPI networks are ideal for real time data visualization. They provide the necessary high speed interface data transfer rates to approach television quality real time display. This resolution requires a 90MB per second data rate (31), to support updating at 24 times a second and 24 bit color.

For this calculation we will assume that our program has exclusive access to an I/O node with a single HiPPI port. The HiPPI system has a nominal bandwidth of 100 MB/sec. The typical transfer rate obtained in practice is around 68MB/sec. These figures were obtained by Messina (53) on a Paragon system. Using the results from (53) we can perform the same calculations using the time required to send data to the HiPPI node and the time required to transfer it over the HiPPI channel to our remote display system. This highlights the delay associated with gathering the data at one node in the Paragon system. The effective bandwidth actually available for real time event animation is reduced from the theoretical value of 100 MByte per second to 6.35 MBytes (see Figure 33). Performing the same calculations as before produces a maximum interesting event tracing rate of:

- Assuming 240 processing nodes and 32 byte events.

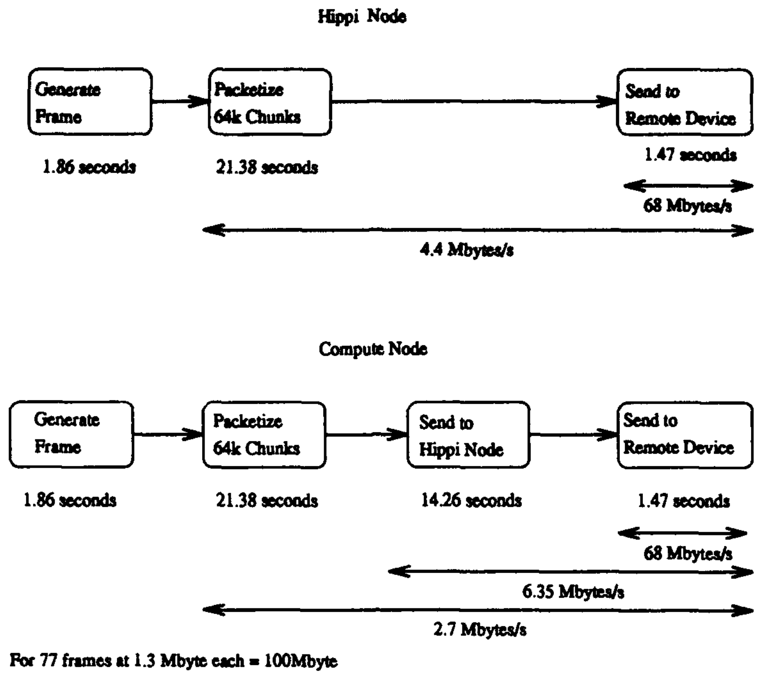


Figure 33. Limitation of I/O Capacity for Distributed Data (HiPPI)

$$Trace\ Volume\ per\ second\ per\ node = \frac{6.35M}{240 \times 32} = 826\ events.$$

In this case however, we begin to place a significant overhead on the processing node interconnection network. While obviously some data reduction can be performed on the nodes, this figure still reflects a good guide to the upper limit of event tracing. In fact, the actual traffic load with ideal message combination would account for less than 5.0% of the channel capacity into the HiPPI node.

7.9.4 Further Data Visualization to Complement Algorithm Animation. With the HiPPI levels of data I/O, visual rendering of partial or transient solutions can be performed. Data visualization is both an important tool for presentation of results and effective technique for examining program execution. With data volume forcing an increase in the level of observation for algorithm animation, greater use of data visualization will be required.

The Paragon now provides significant improvement in graphics capability with increased hosting of graphical applications directly on the Paragon system.

7.10 Graphical Enhancement on the Paragon

The Paragon system takes full advantage of the Unix operating system, which is now hosted on each node. On the previous iPSC systems, only the host processor executed a full Unix system, with the nodes having only a limited communication based kernel NX (76). On the Paragon, each node has a full featured OSF/1 operating system, providing additional file management and process control features. For animation capabilities, the operating system now allows direct access by compute and server nodes to the following features.

- The X Windows System and the Athena widgets.
- Distributed Graphics Library.
- OpenGL graphics system.

This capability allows for the instrumentation system to directly execute on the Paragon system while manipulating the animation on a remote display system. However from my analysis, the advantages of this approach are not clear. The demand on I/O capacity is already the limiting factor and a remote X windows server is not going to improve this restriction. This is particularly important since one trace event can be used to update an unlimited number of animation displays. Only when full use of the HiPPI I/O capacity is available, can significant improvements be made.

7.11 Additional Considerations on the Paragon

The implementation chosen on the Paragon system reflects a trend to more advanced operating systems on the processing nodes. While these changes improve the user's ability to control parallel disk I/O and other system services, it adds an additional layer between the user and the hardware. This separation reduces the user's ability to predict the performance of the system under specific conditions. To animate the execution at a system level, the operating system activity must be captured and animated. This is not a simple task, as it was on the iPSC/2 system since on the Paragon, a program's performance is a function of all user programs currently executing. In particular, the Paragon allows the following operating parameters and functions:

- Different user partitions can be allocated the same processing nodes.
- Time slicing between user applications.
- Unix based priority scheduling of user applications on processing nodes.
- Pass through message routing.
- Horizontal communication channels are used for all disk and user I/O.

The pass through message routing allows messages from different user partitions to be routed through each other's partition. In effect, message latency within any partition will depend on the effect of other applications. As a result, low level communication based displays are unlikely to provide useful information unless sole access to the system can be assured. This problem is further compounded by file I/O. In the Paragon system, disks are allocated to the I/O partition which is physically located at the ends of the horizontal communication paths (Figure ??). Therefore, any application requiring file I/O is likely to utilize the majority of the available bandwidth on these horizontal channels. Since these channels will pass through all but trivially small partitions, the data collected by the instrumentation system would be corrupted. From these operating restrictions, it is clear the low level animation on the Paragon will be dependent on system wide instrumentation.

The current implementation of ParAide does not attempt to address these problems and is unlikely to in the near future. Thus, while ParAide provides algorithm animation capabilities to system users, the usefulness of the resultant data is less accurate than previous animation systems.

7.12 Summary

The analysis in this chapter indicates that the level of observation used during algorithm animation must be considered in order to resolve the problems associated with trace volume. A realistic upper limit on the frequency and size of trace events was developed. For a Paragon system with 240 nodes, real time animation over Internet should aim at less than 200 events per node per second. If a HiPPI interface is used, this can be increased to less than 1000 events per node per second. This can be scaled inversely with the number of processing nodes, however, event combining must be utilized. These values can be used to guide the program instrumentation process. Our algorithm animations should thus aim at depicting events that can be collected at this frequency.

Based on the clear need to animate algorithms at a higher level of abstraction, the next chapter outlines an new direction for AAARF development.

VIII. A new direction for AAARF development

8.1 Changing Requirements for Parallel Animation Systems

From the analysis of trace volume in Section VII and the development cost of application specific displays, it is apparent that a new approach to algorithm animation must be created. This section outlines a new model for future AAARF development. The ideas presented here have been developed based on experience gained while using and developing the AAARF system, from associated literature research and experimental use of other animation systems (Pablo (62), Paragraph (28), AIMS (57), and ParAide (27)).

From the analysis in the previous chapters, it is also readily apparent that parallel algorithm animation systems currently available fall short of real user requirements. This is most noticeable when the current papers are reviewed for examples of effective employment of these systems on real world problems. Without a notable exception, virtually all reported usage of these systems is by their development team on pedagogical examples. While from an educational perspective these are valid and important applications, it is vital that these tools also support main stream software development. It has been made apparent from Parallel Application and Algorithm Group (PAAG) meeting discussions that researchers at other Wright Patterson AFB facilities make virtually no use of these types of systems. There would appear to be four major problems with current animation systems that lead to this low usage:

- *An inappropriate level of observation for observing algorithm level characteristics.* Apart from the basic problems of trace data volumes, the system level event tracing is actually not as applicable to the development cycle as first envisaged. Observing generic animations for a particular execution is effectively the same as watching the instruction flow without seeing the data on which it is based. As a result, the reasons for particular behavior cannot be evaluated by the user. This forces development of application specific displays for virtually all serious applications.

- *The inability to debug non terminating or incorrectly terminating programs.* Nearly all current animation systems require the program to gracefully terminate to allow the data collection system to finalize collection of recorded event data. Since they provide no execution control of the program under examination, finding implementation errors is virtually impossible. A debugger such as IPD (38), the interactive parallel debugger, is far more appropriate for this application.

- *Difficult to apply to real optimization activities.* Optimization of a program's execution is often cited as an application of animation systems. While useful from a pedagogical perspective, it is ineffective on real problems. Most research in parallel computing today requires that this optimization activity focus on register and memory allocation, file I/O, and hand tuning of program kernels. With standard routines for message passing operations such as exchanges, the level of observation provided by animation systems is too coarse. In particular, when the level of observation is reduced to examine these parameters, the intrusion caused by the instrumentation becomes unacceptable.

- *Complexity of trace collection and animation system.* Since to gain any real benefit from animating a user's program, he/she must employ application specific displays, the work required is significantly increased. In virtually all systems currently available (the only notable exception is Pablo) this requires a significant amount of programming. Faced with this prospect, it is not surprising that it is not often pursued by the user.

Thus for real benefits to be gained from animation systems, high level algorithm displays must be generated. However, current systems focus mainly on low level system events. This leads us to two basic questions for current animation systems:

1. *How do we reduce trace volume?* This would appear to force the development of high level algorithm animation with composite type events, relegating current visualizations to limited debugging and optimization of very small code segments in an educational type environment.

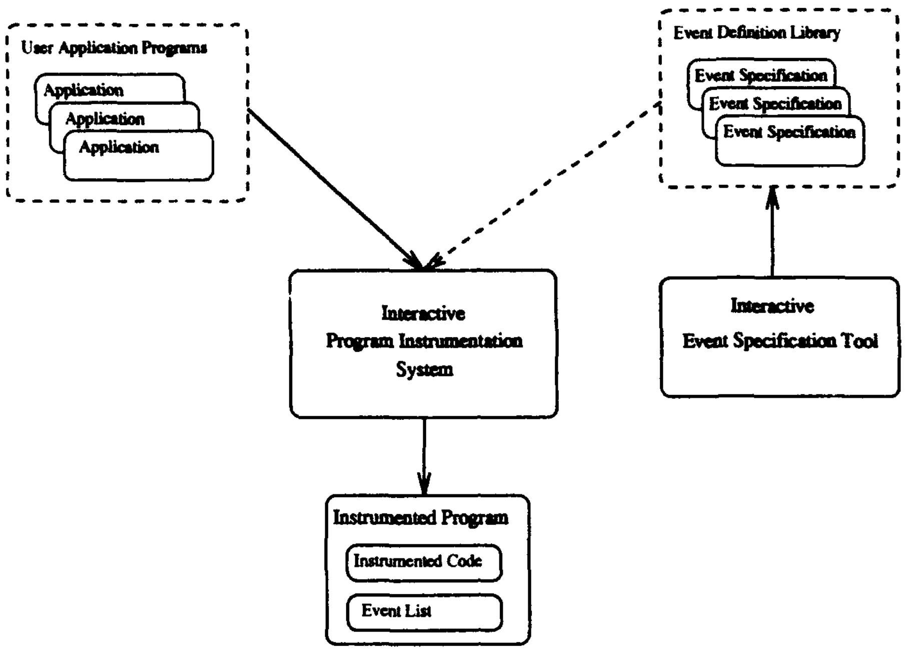


Figure 34. Proposed Instrumentation Environment

2. ***How do we eliminate the need for lengthy development of application specific animation for high level algorithm events without loss of capability?*** This directs research into the development of interactive tools/environments for animation building rather than the development of specific displays. This is more in line with standard data visualization such as Explorer (75).

These two points provide the focus for the proposed animation environment model. The following section outlines the major components of the proposed system. The section discusses both the overall philosophy behind the development and the importance of the individual components. A block diagram for the proposed system is shown in Figures 34, 35 & 36. The proposed model includes the following important characteristics:

- Architecturally independent self defining event tracing.
- Top down hierarchical trace resolution.
- User defined and configurable event filtering.

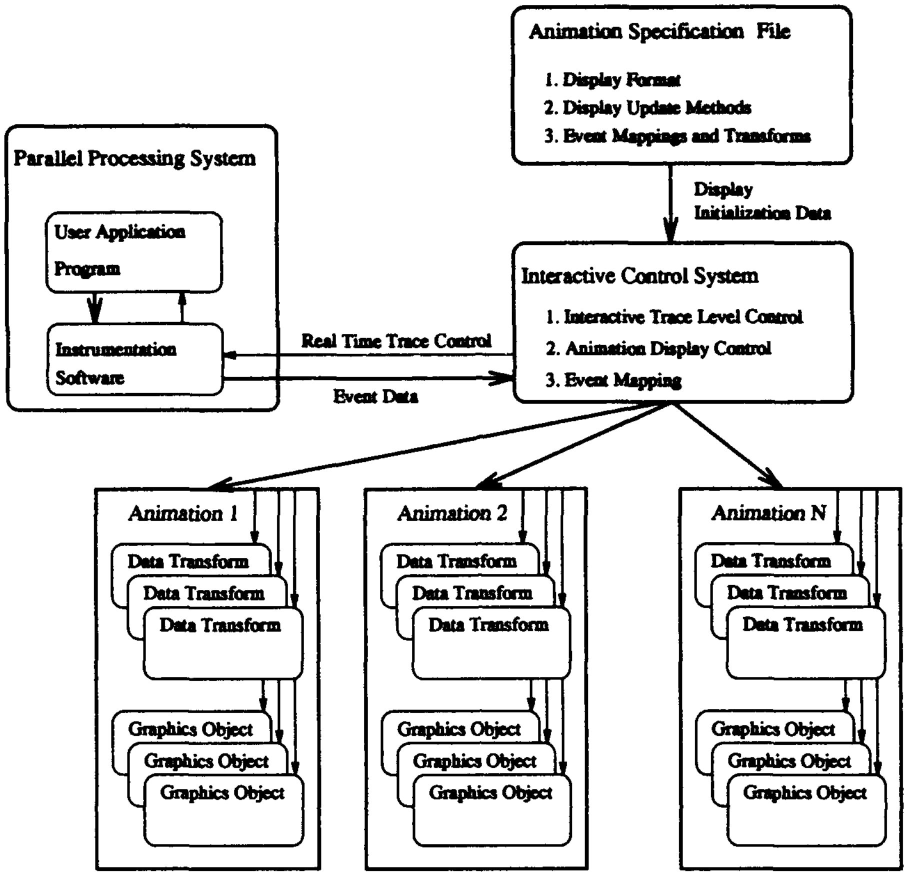

Animation Specification File
1. Display Format
2. Display Update Methods
3. Event Mappings and Transforms
Parallel Processing System
User Application Program
Instrumentation Software
Display Initialization Data
Interactive Control System
1. Interactive Trace Level Control
2. Animation Display Control
3. Event Mapping
Real Time Trace Control
Event Data
Animation 1
Animation 2
Animation N
Data Transform
Graphics Object

Figure 35. Proposed Real Time Animation Environment

- Integrated multidimensional data visualizer.
- User specified interactive graphical display builder.
- Pseudo real time display presentation.
- Pseudo real time user manipulation of algorithms including:
  - event tracing level.
  - data structure values.
- Portable animations on high performance graphical workstations.

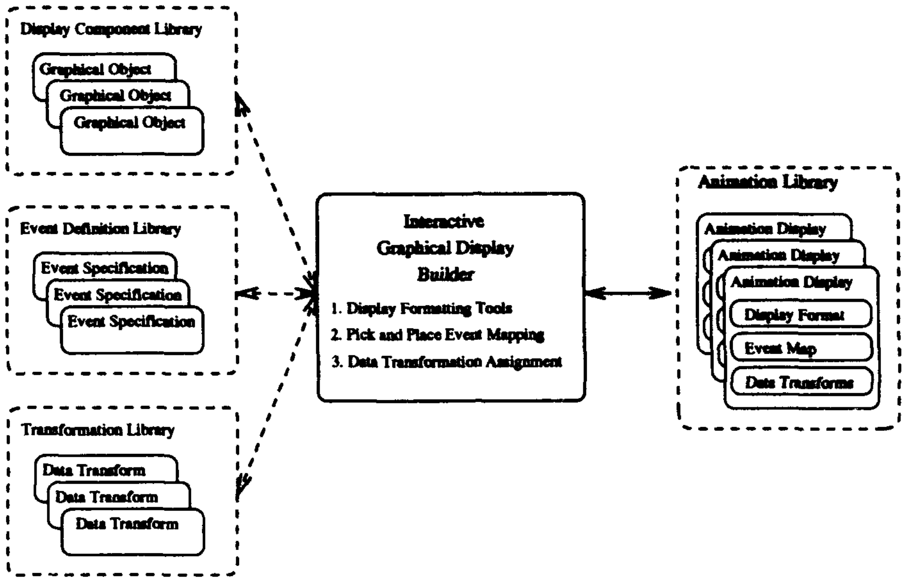


Figure 36. Proposed Animation Development Environment

8.2 Architecturally Independent Self Defining Event Tracing

Animation requirements have tended to vary from user to user and application to application, as a vast array of event types have been defined for parallel event tracing. While some of these events remain relatively static, i.e message passing, the content of higher level trace markers vary considerably. Even the flexible event markers available in AAARF (84) were restrictive in that mixed *type* information could not be stored as a single event. In many other systems the size of events is limited. This has led to obsolescence in event formats as the needs of the user have changed over time. The only effective solutions to this problem are to implement a self defining event format, or basic events that can be grouped during analysis. These approaches restrict neither the event contents or size, while still allowing unique identification.

The main advantage of this approach is the ability of the user to create new events based on his particular requirements. This is ideal for the development of application specific animations since it can capture the contents of complex data structures used by the program under examination. This has been the successful approach adopted by the Pablo system and subsequently included into the Paragon (39) OSF operating system. While significant scope for improvement in the actual implementation of the trace format is possible, its adoption by Intel creates a pseudo standard. With AFIT's likely strong relationship with the Paragon system, our future developments must take advantage of the Pablo event format.

It is very likely that the user of an animation system would be daunted by a completely flexible event format and the semantics of its construction. The subsequent decoding of its structure and mapping of data values to objects on the animation displays would also be difficult. To alleviate this problem while allowing for the flexibility of composite events, the prototype implementation outlined in Chapter IX uses single valued events. The user simply inserts the required number of single valued events to construct a composite event. The additional cost with this approach is that tracing overhead is increased along with the requirement for storage of additional time stamps.

8.3 Hierarchical Trace Resolution

The problem of trace data volume continues to limit the detail that can be displayed for any reasonable execution period. This does not however, eliminate the need for low level system event tracing in some situations. To provide the best of both requirements, the event tracing must be arranged in layers so that the user is able to vary the level of detail. For example, an abstract display for a parallel discrete event simulation may provide a run time display showing only the lowest time stamped message in the system. If however, the simulation appears to not be progressing, the trace level could be decreased to depict the contents of the individual messages. For the reason outlined in Chapter VII, this must actually be the enabling of detailed tracing, not simply the presentation of previously un-displayed data. To always have the tracing system enabled would place an intolerable overhead on execution. This is often referred to as a top down approach to instrumentation. In the top down approach, we only collect what is needed at that particular level and increase trace detail only when required. The alternative approach used by nearly all current systems is the bottom up technique. With this technique, a detailed event stream is filtered and events combined to provide high level displays.

Hierarchical trace resolution could be achieved by allowing the user to establish ordered sets of event classes and as events are inserted into the program, or created by the user, they are assigned to these classes. The user would then be provided with selections that allow him to zoom in and out on the event resolution, by enabling or disabling event classes. This system could be as flexible as required with low and high level events being combined into particular classes to meet specific requirements.

In addition to implementing hierarchical event classes, the individual events would be assigned a collection frequency. This frequency could be determined from program parameters, or a simple counter, to ensure that data structures that experience frequent updating do not produce excessive event data. An interesting alternative would be to allow the user to select a specific level of tracing overhead and allow the system to adapt to maintain this level. In this situation, the instrumentation system could track trace data volume at the node and reduce the reporting rate to meet requirements.

8.3.1 Self Adapting Event Tracing. It would appear to be a desirable property to allow the tracing system to adapt to user requirements by self adjusting the trace level. It is often difficult to determine the rate at which particular data will be generated and in some cases, this rate can vary greatly during different phases of execution. Particularly in cases where dynamically allocated data structures or object based programming techniques are utilized, the actual volume of data required to capture a particular event can vary dramatically. In these cases, a self adapting system could ensure that data generated never exceeds some fixed ceiling. A particular node could store the average number of trace data bytes per unit time and modify the trace capture frequency accordingly to meet user specified requirements.

8.4 User Defined and Configurable Event Filtering and Transformation

While it is possible to produce a variety of displays capable of depicting every event, this is not always the most usable format for the data. This approach has lead to current systems including extensive menus that allow different aspects of events to be examined. This approach was reasonable when events were strongly typed, however, this method is poorly suited to handling flexible event types. To compensate for this and to provide user flexibility, lacking in many present animation systems, an interactive event filter capability is proposed. This technique has been used in the AIMS (57) system to great effect.

The approach proposed would require a user interface that allowed the building of filter operations in an interactive manner. The event types generated by the user would be presented in a pull down menu and, using techniques often used to specify data base searches, a filter could be composed. The filter would not simply be limited to selecting particular events but should allow combination, min and max detection, averaging and other statistical techniques, translation, scaling, bound checking and other appropriate operators. The resulting code module would then accept events in real time and pass them to one of the many display objects.

A good example of this concept is the Explorer data visual system (75). This system provides a selection of graphically composible objects that can be used to produce a specific data transformation for data visualization. The development of a pick and place type environment would be an essential feature.

8.5 Integrated Multidimension Data Visualizer

As the complexity of programs in daily usage continues to increase, determination of progress and state parameters becomes more difficult. This is particularly true when algorithms manipulate data that relates to some multi-dimensional problem space. In these situations, visualization of partial results becomes increasingly important. This can be particularly important in optimization activities, simulation and modeling of real world phenomena. It is thus important to provide as part of the visualization environment, an interface to a state of the art data visualization tool.

Recent discussion by researchers from Oak Ridge National Laboratories and other research organizations, at the Intel Supercomputer meeting (36) highlighted the lack of adequate tools in this area. The problem is not directly related to tools that can generate the displays, but the utilities required to gather the data for the presentation system. In the MIMD type machine examined as part of this research, the data to be visualized is often scattered across multiple processing nodes. Currently only limited data gathering utilities are available for construction of multidimensional displays on parallel processing systems.

The opportunity exists for the self defining trace events to be used to define events that can gather distributed display data. The events containing the data could be decoded using the user configurable filtering system and piped directly to the visualization tool. The visualization tool would not be constrained to any particular type since the adaptable filtering could produce data formats compatible with a range of visualization systems. Three dimensional diagrams depicting partial solutions or system progress could then be constructed and adapted to meet user monitoring requirements.

This technique is also useful when analyzing lower level statistical information. The most relevant would be message volume distribution. It is relatively easy to simply count the messages on each communication channel without recording any specific time or semantic information. The distribution could then be displayed on a two dimensional grid as implemented in the AIMS system. The grid type displays in AIMS are generated by Explorer (75).

8.6 User Specified Interactive Graphical Display Builder

The most limiting aspect of AAARF and a problem also inherent in the AIMS (57), PICL (28) and other systems is the lack of flexibility in display format. Each variation in display format

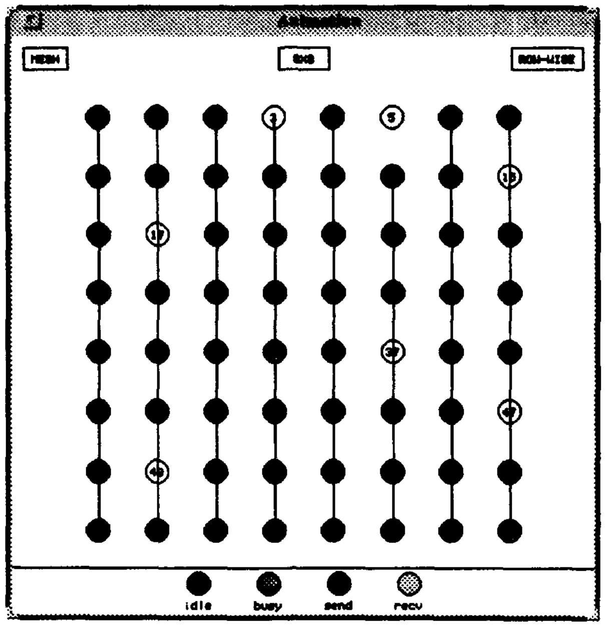


Figure 37. Mesh Processor Animation from PICL

must be implemented by directly writing graphical calls to an Xwindows widget set. While widget support is available for the development of user interfaces using panel, buttons, menus etc, similar libraries are not well suited to algorithm animation.

A classic illustration of this is when producing a simple activity display for processor nodes. For example, we may want to represent each processor as a circle on the display with the color representing a variable on that processor or simply the processor state (active/idle). The animation display shows a similar type display with the processor displayed in a ring (see Figure 38). In this case, the color only reflects activity. If the processors are actually configured as a mesh code for a completely new display, procedures must be developed as shown in Figure 37. The menu buttons for its selection must be added to the control panel as with any required parameter such as aspect ratio. This approach is implemented in PICL (28) and as a result they have coded 10 different arrangements of processors (see Figures 38 & 37). This however, still leaves the problem of monitoring a different parameter not simply idle or active. In this case, the code section that generates the display must be duplicated and the displays modified to accept a different event type

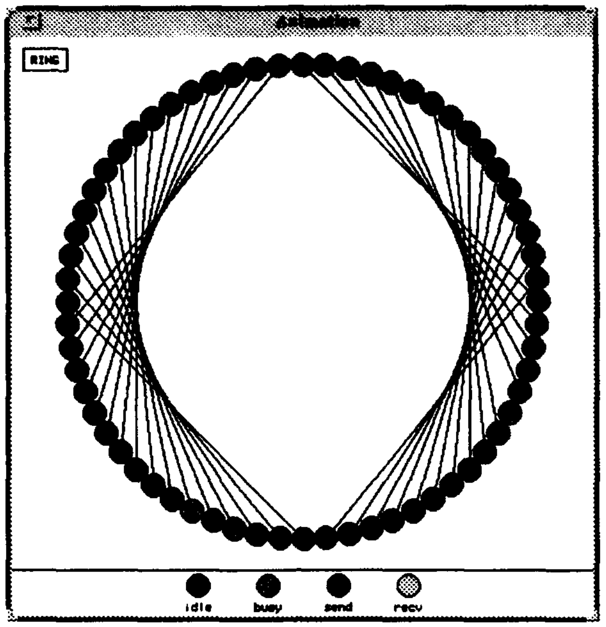


Figure 38. Ring Processor Animation from PICL

and color coding key. The limitations of this approach are painfully apparent. The problem does not end here however, as frequently further abstraction is required.

When a problem is decomposed onto a parallel system, the actual activity of a large number of nodes becomes difficult to determine. Clustering of related activities and similarity of tasks often results in the individual activity of each processor becoming less important to visualize. Thus it is frequently desirable and often essential on large systems, to abstract the behavior of individual processing elements. Under this situation, the max, min, average or some other summary form can be used to represent the activity of a group of processors. At the same time, particular nodes may warrant individual attention. Thus, while we can produce displays that represent the processors in arbitrary configurations, this is not flexible enough for the majority of applications. This is particularly relevant as the number of processors increases. For example, a search conducted using 2800 node on a Paragon system using 50 nodes in tree type control structure with the remaining nodes acting as workers, would not fit a standard display format easily. Even if it was coding a particular, display using a template type approach would still be a significant task.

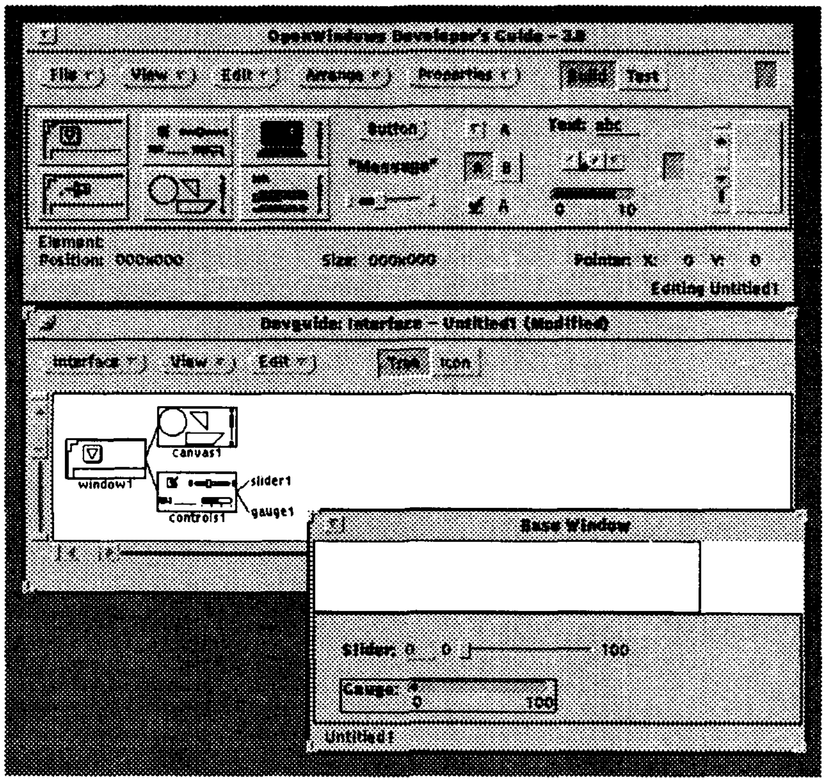


Figure 39. Interactive Interface Builder for Xwindows

This problem of presentation format is inherent in all displays examined by this research. The most logical solution to the problem is to allow the user to configure the display without the need for actual code development. This approach is frequently used in developing user interfaces. Under Xview, the Open-windows Developer's Guide version 3.0 (80) allows the user to select interactive components from a graphical menu and compose them at will. This is depicted in Figure 39. The code required to actually implement the user interface is generated automatically. This concept is directly applicable to developing animation displays. In this way we provide the user with tools to build specific display instances rather than providing a set of fixed format displays.

8.6.1 Animation Environment. The following items further outline the desirable characteristics of the proposed environment.

1. *High Level Interface.* The animation tool must allow a pick and place type user interface with a palette of standard animation primitives. A frame representing the final display would be manipulated by the user to add basic animation objects and reconfigure their location.

2. *Animation Objects.* The system should contain a library of basic building blocks such as scrolling bar, colored circles, interconnecting lines and other frequently used graphical objects applicable to algorithm animation. The user must also be able to generate new basic components.

3. *Assignment of State.* Each defining parameter of the animation objects must be accessible and definable to the user. For example, parameters such as size, location and color must be controllable, then allowable states for the characteristics of the graphical objects could be defined. For example, a line could be visible or hidden.

4. *Assignment of Events.* Within this environment the event types would be interactively assigned to defined graphical objects. The data fields in the event would be presented to the user along with range boundaries. The individual fields in the events would be assigned to a particular parameter of one, or many graphical objects. A simple example would be a particular event containing an integer array which could be assigned to a display containing a bar graph animation. The events could also be used to initialize or highlight particular display format.

5. *Animation Library.* Once the displays have been developed they could be stored in a domain specific library, with the definitions of any unique event types for possible reuse.

The basic ideas presented in this section have been incorporated into an experimental prototype outlined in Chapter IX. While this has been incorporated into AAARF to facilitate rapid development, the code has been developed to allow construction of a replacement framework for AAARF components of the design.

8.7 Pseudo Real Time Display Presentation

As discussed previously, the input/output capacity of parallel systems is likely to remain a serious constraint. Any improvement could be easily consumed by the requirements of large applications, the majority of which are I/O bound for disk access outside node network. As a result, the detailed low-level real-time event tracing provided by AAARF is likely to consume too high a percentage of this resource. This does not however, limit tracing of events that produce lower data volume.

For most large applications only high level event tracing is likely to be possible. Under these situations, the number of events generated can be significantly limited. For example, a particular program variable could be reported once every ten minutes in extended execution with only limited impact. For this reason, pseudo real-time animation remains an important feature of the new animation system. The most important application area is related to long execution time where it can be used to determine:

- if a program is still executing,
- if progress is being made,
- the quality of partial results,
- the execution pattern, and
- its compliance with execution constraints.

These aspects are becoming increasingly important as larger problems are attempted, consuming hundreds of computer hours. Under this scenario, the early detection of problems that do not cause termination, but produce invalid solutions or infinite execution is vital. The information presented could be as complex as three dimensional rendering of images or as simple as basic bar charts. This is often implemented with programs dumping partial results to file for subsequent display. While this is sufficient for many applications, the flexibility provided by user interaction outlined in Section 8.8 provides some further advantages not possible with this approach.

8.8 Pseudo Real Time User Manipulation of Algorithm

With the majority of parallel animation systems, tracing level and execution parameters are determined prior to execution. Once execution has commenced, these cannot be altered until the program is rerun and often must be recompiled. AAARF and PICL require this for some trace parameters.

If the pseudo real time displays are used effectively, insight into the execution of the program might highlight aspects of the execution pattern not previously considered. In addition the progress made by the program might highlight an appropriate change in a data value. An example of this might be mutation rate in a genetic algorithm program. For short executions the program could simply be rerun but on larger systems with huge execution times, this may be ineffective. The approach proposed here would be applicable to programs where the cost of re-running the program completely would be prohibitive. Under these situations, the ability to change both the trace level and particular data values or structure would be an advantage. While the technique of *check pointing* a program's execution is often used to allow restart, it has limitations. Often the information required by a parallel application consumes a high percentage of the total available storage capacity. In these cases, frequent check pointing is not possible due to the amount of data that must be stored to retain the system state. In this situation, visualization can be used to determine if the current program state is valid and the previous checkpoint replaced by the current state.

To allow user input, the node processor must be capable of receiving messages from the display system. The instrumentation software would need to include additional code to check for messages from the display host. All defined events, either currently enabled or not, would check incoming message queues for any message relevant to themselves. This could be implemented using a reserved message type. The trace events could in this way be enabled, if later in the execution it was required to investigate unexpected behavior. A hard disable could also be included or a checking frequency defined to reduce the effects of the resulting computational overhead. In this case, data that was instrumented but too costly to output to the display device could, from time to time, be enabled if further analysis was required. A similar approach could be implemented to allow modification of data structures and values. The self defining event type would ensure correct casting of both the sent and received data.

This would be the most unique aspect of the proposed system and would require co-operation from vendors to allow enhancement of the tracing code. The following pseudo code for an event tracing routine outlines the concept.

Trace Event Code Segment.

```
Procedure Event'Marker(Event'Type, Trace'Parameters)
 static CHECK'ITERATION, RECORD'ITERATION: Integer;
 Begin
 if not HARD'DISABLED do
  Begin
   CHECK'ITERATION:= CHECK'ITERATION + 1;
   if ITERATION=CHECK'FREQUENCY and Probe(COMMAND'MESSAGE)=TRUE
    Begin
     Switch COMMAND'MESSAGE.type
      Set'Disable:
          ENABLE := False  ;
      Set'Enable:
          ENABLE :=True ;
      Set'Hard'disable:
          HARD'DISABLED := True;
      Set'Check'frequency:
          CHECK'FREQUENCY := COMMAND'MESSAGE.data'value;
      Set'data'value:
          Trace'Parameters := COMMAND'MESSAGE.data'record;
      Set'record'frequency:
            RECORD'FREQUENCY := COMMAND'MESSAGE.data'value;
        Set'terminate:
            TERMINATE := True;
      End
      ITERATION := 0;
    endif
    RECORD'ITERATION:= RECORD'ITERATION + 1;
    if ENABLE :=True and RECORD'ITERATION = RECORD'FREQUENCY
      Begin
       Buffer(Event'Type, Trace'Parameters);
       if Buffer = FULL or TERMINATE = True
        Begin
          Send(Buffer) to I/O processor;
        End
       RECORD'ITERATION:= 0;
      End
     endif
  End Event'Marker;
```

8.9 Portable Animations on High Performance Graphical Workstations

As both the speed and size of parallel processing systems increases, the requirement for higher resolution and display update rate increases. In particular, the animation speed of the current AAARF system is dictated by the graphics capability of the Sun workstation. For this work to continue, use of higher performance display systems is required. In the case of current AFIT research, the graphical capabilities of the new Silicon Graphics machine should be utilized.

8.10 Can this be Achieved by Further Upgrading of the AAARF System?

Considering these requirements, the basic framework of AAARF has reached the limits of its extendibility. While the concept of user configurable, or adaptive animations was investigated within the AAARF framework, AAARF cannot support incorporation of all proposals. Continued developmen' of the current AAARF system would require at least the following items to achieve the additional requirements outlined in Section 8.1:

1. Porting of the system to the Silicon Graphics machine.
2. The complete rewrite of the unsupported SunView (80, 79) graphical display code.
3. The conversion of the XView (30, 61) menu framework to be Motif compliant (56).
4. Complete replacement of the trace data collection system and subsequent processing format.

This constitutes a significant undertaking especially as the third task alone required a complete thesis cycle when originally converted from SunView to XView (86). This is not however, the major reason for replacing AAARF with a second generation animation system. The AAARF system has two characteristics that, while shared by the majority of current animation systems, are incomparable with the proposed direction of future research. Firstly, its animation capabilities are directly tied to the data casting of the Prase data collection software. This forces every event to be cast as an identical record type irrespective of the information required. With future research beginning to focus on the Intel Paragon (39) the move to self defining event types will be required, making the single event record type redundant. Secondly, AAARF's code based animation building technique is poorly suited to adaption to an interactive graphical display builder type system. The AAARF system was designed to associate animations with a particular algorithm class. Within this framework, a unique control panel, event decoder, display menu. status panel, and presentation parameters panel must be built. This requires the customizing of five application specific AAARF program files before even the first animation is constructed. While the work on adaptable displays discussed in Chapter IX was developed within this framework, it remains too restrictive for full scale development of interactive animation building tools.

8.11 Summary

A model for future parallel algorithm research at AFIT has been developed. It cannot easily be implemented within the framework of the current AAARF implementation. To take full advantage of progress made by commercial organizations and other researchers in this area, the development of a new generation of visualization systems must be undertaken. The rapid progress made in the development of parallel processing systems changes many of the underlying principles on which the previous systems were based. Processor speed and the sheer magnitude of new systems continue to challenge the developers of animation systems. The concept of user definable displays is further developed in Chapter IX.

IX. Adaptable Displays for Application Specific Animations

9.1 Introduction

This chapter outlines experimentation with adaptable display configurations for application specific animations. The prototype system developed here reduces parallel algorithm animation to its most basic components. By assuming the use of a single event type, the user can build application specific displays without requiring code generation. In particular, the user is able to develop arbitrary display formats to meet any animation requirement. This work has been undertaken within the basic framework of the AAARF system. For development of application specific displays with both AIMS, and Paragraph the user must modify components of both the instrumentation and the graphical display software.

9.2 Design Concept

The basic approach employed by this design is to provide configurable animation displays that rely on: a set of graphic object primitives (frame); a single event format; an event map; and set of a statistical operations. This approach provides the maximum flexibility and imposes only trivial requirements on data collection formats. A single event type has been utilized to both simplify implementation and provide users with less complex instrumentation procedures.

The system operates by the user specifying a frame format based on a set of graphical objects. For each of the parameters associated with that graphical object, the user assigns its initial values. This defines the static configuration of that display. The user then maps individual event types to particular parameters, or sets of parameters on one or more graphical objects. For each mapping the user then defines a particular operation type to be applied when updating that particular parameter. For example, the actual data value can be used directly or transformed by scaling, averaging, summed, counted, max, or other mathematical operations. For each frame of objects, the manner in which it is to be updated is also defined to allow scrolling and other operations on

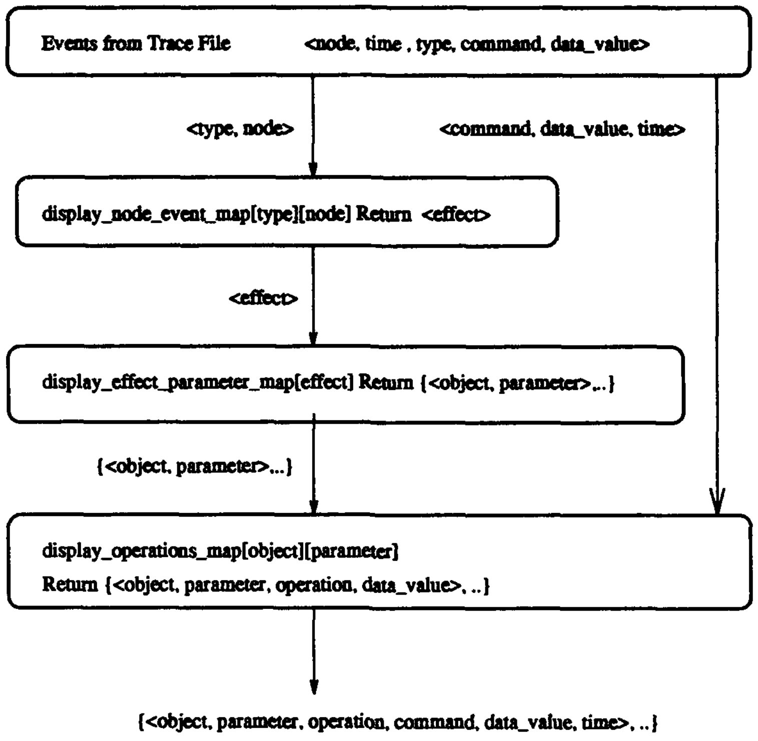

Events from Trace File
<node, time , type, command, data_value>
<type, node>
<command, data_value, time>
display_node_event_map[type][node] Return <effect>
<effect>
display_effect_parameter_map[effect] Return {<object, parameter>,..}
{<object, parameter>...}
display_operations_map[object][parameter]
Return {<object, parameter, operation, data_value>, ..}
{<object, parameter, operation, command, data_value, time>, ..}

Figure 40. Mapping of Event Data to display Object Parameters

object sets. The event mapping procedure is depicted in Figure 40. A block diagram of the adaptive display environment is shown in Figure 41. This figure clearly shows how the display definition file specifies the graphical animation. The additional maps, methods, and transforms define the effect incoming events will have on the display.

9.2.1 Event Types. The use of only a single event type actually provided significant benefits in both implementation and ease of use. The AAARF *IE(type,data1,data2)* event type was used to allow collection of trace data. No other event types are utilized for adaptable display frames. Other event types can, of course, be inserted to provide data for other animations. The actual user format for the event is simply:

$$IE((int)TYPE, (boolean)COMMAND, (float)data\_value)$$

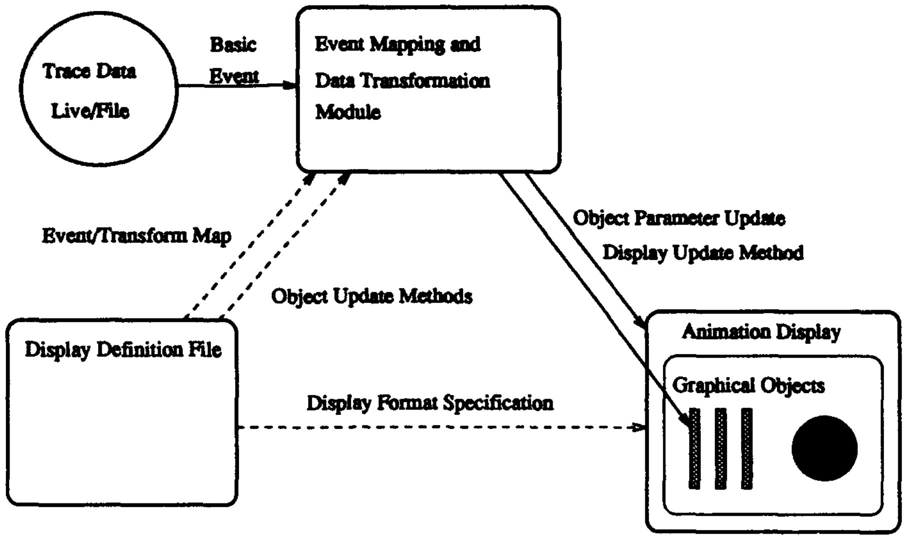


Figure 41. Overview of the Adaptive Display Generation System

The *TYPE* is used along with the node id number to uniquely identify the event to the event map. The *data_value* is then used to update the parameters to which it is mapped (note, an operation on the *data_value* can be specified). The *COMMAND* field (boolean) is used to specify that this event should cause the display to update the graphical object. This allows a number of object parameters from a set of events to be updated as a single action.

9.2.2 Graphical Objects. This implementation provides basic graphical objects for display including, Dots, Rectangles, Text Messages, Circles, and Lines. However, since these can be composed in any manner and stored for reuse, definitions of complex composites can be easily constructed. An object based approach was taken in this implementation and the user has access to the attributes shown in Figure 42. Note: The array of *scale*, *log scale*, and *offset* values provide static translation of the object's attributes.

9.2.3 Frame Specifications. The frame specification allows the users to specify how the objects on the display are actually updated. The options available are:

- S_NONE: All updates and written directly to the display.

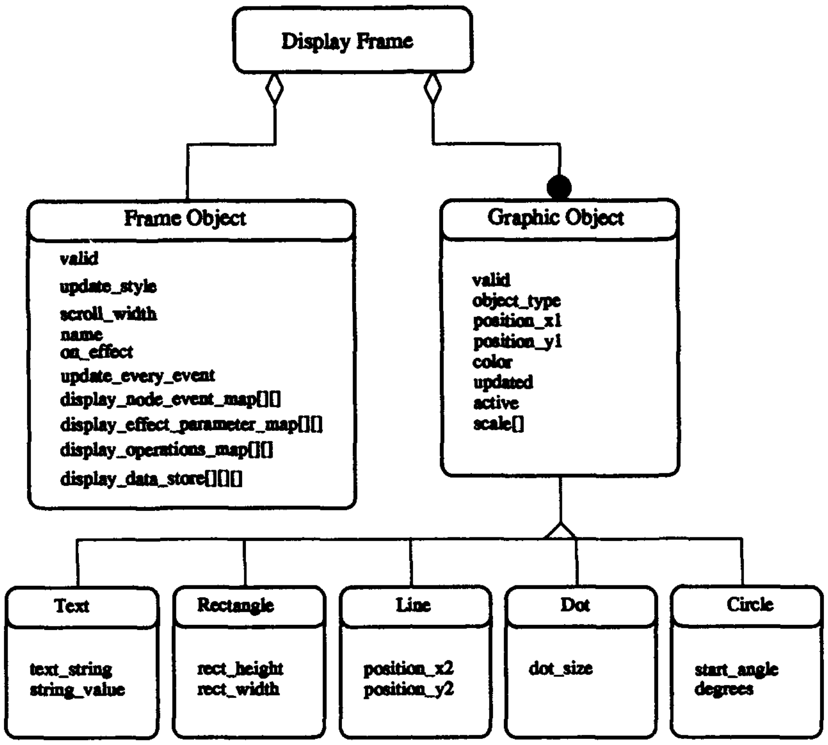


Figure 42. Object Diagram for Adaptive Display Frames

- S_FULL: Clears previous display and redraws all valid objects.
- S_SCROLL_LEFT_ON_TIME: Scrolls, to the left, a user defined percentage of the display frame, based on specified duration.
- S_SCROLL_DOWN_ON_TIME: Scrolls, down , a user defined percentage of the display frame, based on specified duration.
- S_PARTIAL: When updating display only erases and redraws updated graphical objects.
- S_UPDATE_ON_EFFECT: Only redraws the display on arrival of a specified event.
- S_CLEAR_ON_EFFECT: Clears the old display on the arrival of a specified event.
- S_SCROLL_LEFT_ON_EFFECT: Scrolls, to the left, a user defined percentage of the display frame, on arrival of a specified event.

- **S_SCROLL_DOWN_ON_EFFECT: Scrolls, down a user defined percentage of the display frame, on arrival of a specified event.**

Full updating simply clears the last display and re-draws all valid and active display objects associated with that frame. If partial updating is specified then only display objects that have a valid updated flag are erased and redrawn. The COMMAND flag is used to set the ***update*** object flag when a parameter is updated. Thus, to use block updates PARTIAL mode must be used. The scrolling options operate with a scrolling width parameter that is also associated with the frame. Under this method a section of the display (specified by the scroll width) is translated across the display in the direction of the scroll. Under this approach graphical objects are specified for only part of the display and their previous values are shown in the remainder of the display. The scrolling action can be triggered based on execution time, or on the arrival of a particular event. The update on event option clears the current view on arrival of a particular user defined event. If the no updating option is selected, new images are drawn over the previous display. Further information is contained in the AAARF User' Guide (58)

9.2.4 Event Mapping. A two stage mapping o[illegible]nts to display objects and parameters has been implemented. The mapping process is depicted in Figure 40. When an interesting event (IE) is processed by the animation system, the event type and recording node id are used to determine an effect number for that event. Utilizing both the node id and the event type allows events from different nodes to be mapped to the different objects. The effect number is then mapped to a set of object parameter pairs. These pairs represent particular objects and parameters to be updated with the data value from that event. The object parameter pair is also used as the index into the operations map. To allow a number of parameters to be updated before the object is actually redrawn on the display, an updated flag is included in the object structure. The COMMAND value is loaded into the updated flag when a parameter is actually updated by an event.

9.2.5 Operations Mapping. Mapping events to object and parameter pairs is only the initial stage in updating a particular object on a display. The way in which the actual value stored in the received event effects the object can be defined by the user. For example, it could be used directly to map to the height of a rectangular display object, or the height could simply reflect the average value of all events received. Note: Scale and offset factors are automatically associated and applied to each parameter. The direct approach is often very useful but additional capabilities are required. The operations map allows the user to assign to an object parameter pair, the method to be used to update that particular parameter. The operations provided by this implementation are:

- Direct - Updates the parameter directly with the scaled and translated data value.
- Max - Updates parameter directly with the scaled and translated data value if it was greater than the largest previously received value.
- Min - Same as max but retains only the minimum value received.
- Count - Updates the parameter with the number of that type of event received to date.
- Sum - This operation updates the parameter with the summation of all event data values received.
- Average_Time - Updates the display with the average value of data_values received over a specified time.
- Count_Time - Updates the parameter with the number of that type of event received over a specified time period.
- Reset_on_event - This operation allows the arrival of an event to clear the current values of count, sum, and average. The current display value is maintained at its last value before the reset operation.

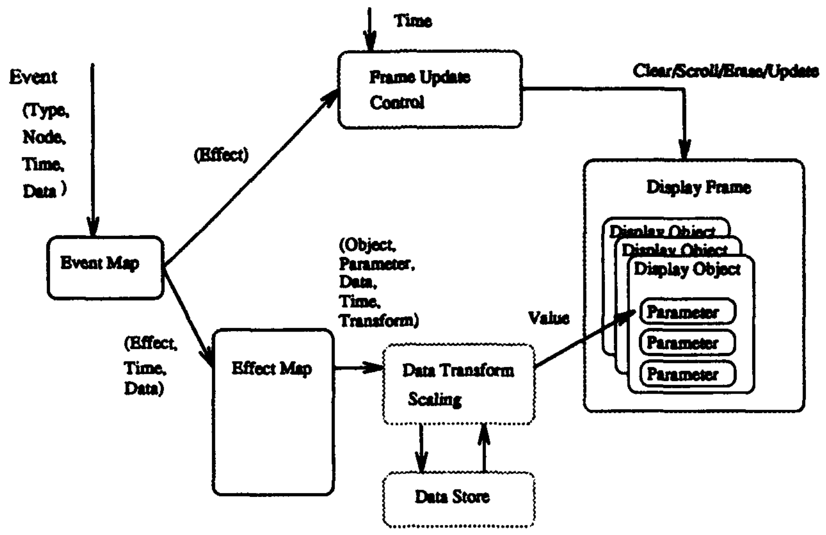


Figure 43. Event Processing and Display Control

- Reset_to_zero_on_event - This operation also clears the current values of count, sum, and average. This particular operation resets the current data value to zero.
- Time_Events - This operation allows the time between occurrences of events to be mapped to a display object.

The animation system uses the array display_data_store[object][parameter] to store data required to allow evaluation of these functions (Figure 43).

9.2.6 Specification File. The adaptive displays are constructed by the user specifying the required objects in an ASCII file. The file can contain the definitions of up to four different display frames. This upper limit is not fixed and simply reflects the current number of programmable menu buttons on the AAARF control panel allocated to adaptable displays. The file is normally read only at initialization of the adaptive display class. At any time a new definition file can be specified and subsequently loaded by selecting *Reset.* As the file is read, the frames assigned by the users to the particular display formats are allocated to menu buttons on the control panel to allow particular frames to be displayed. A discussion and specification of the file format is contained in

Appendix A. Users of the system should consult the AAARF User's Guide (58) for more detailed information.

9.3 Applicability of Approach

The approach in this chapter provided some promising results. The speed at which new displays could be constructed, even without an interactive interface, was most encouraging. For example, the simple state based animations could be constructed in under two hours. The major strength with this approach is the single event type. This not only reduces the complexity of the user design task, but also makes the instrumentation process more straight forward. This is achieved without limiting the construction of more complex events from these basic forms. From this work it became apparent that basic building blocks provided here could be used to construct an equivalent display to every format so far encountered with parallel animation systems. Achieving this without coding and by utilizing only a single event format is a significant reduction in animation construction complexity.

The approach taken is similar to that of Pablo. Under Pablo the user is supplied with a number of fixed display formats: i.e Bar Graph, Chart, Dial, and many others. The user then defines how complex event types are to be mapped to the variable characteristics of that particular display. This functionality is then aimed at examining low level and system performance aspects of the program's execution, for example, average MIPs values per processor. Development of other abstract displays requires coding. Under the approach outlined in this chapter, we allow greater flexibility in display formats without resorting to coding. The event data we aim to capture, in real time if required, would focus on the algorithm performance and less on system parameters. This level of observation requires more specialized displays built for particular application programs. Thus our design benefits from allowing specification of both the display format and the event mapping without coding.

Since our displays are built on a very simple event type, our data collection requirements are independent of specific hardware or software architectures. This should allow analysis of data from diverse sources, ensuring portability. This prototype implementation does not include interactive interfaces and other components required for easy user construction of displays and event maps, however, it does provide a basis for such work.

9.4 Summary

This chapter has outlined a versatile approach to application-specific animation generation. This approach frees the user from many of the problems with current visualization systems and appears to provide flexibility to meet most animation requirements. In particular, we are able to construct complex application specific animations without requiring code development. The requirements generated by parallel program design and system hardware are changing too rapidly for fixed format display, no matter how scalable. The regularity inherent in the majority of parallel program implementations must be utilized to provide abstraction of uninteresting detail. This can only be achieved by providing the user with the ability to construct displays tailored to his particular application. The flexibility provided here meets this requirement by ensuring that the animation system software remains independent of both the hardware system and the software under examination. In this design the development of application specific displays and event formats can be achieved without code generation.

The following chapter demonstrates how this environment can be utilized to meet real animation objectives. In particular, the animation environment is applied to generation of application specific animation for evolutionary algorithms.

X. *Animation of Evolutionary Algorithms*

10.1 Introduction

Evolutionary Algorithms (EA) are powerful and widely applicable search and optimization techniques based on principles from evolutionary theory (54). There exist many variations of the EV principle and these include, Genetic Algorithms (GA), Evolutionary Strategies (ES), Evolutionary Programming (EP), and Genetic Programming (GP) (54, 32, 48). All of these techniques can be viewed as variations on a basic heuristic approach that share common implementation mechanics.

This chapter specifically examines the development of visualizations that directly support GAs (26). However, the similarities between various types of EAs allow the basic techniques developed here to be applied to all GPs, ESs, and, EPs. Use of the prototype adaptive animation extensions to AAARF, discussed in Chapter IX have been made. This section clearly demonstrates how effective animations can be developed using this adaptive animation environment.

10.2 Genetic Algorithms

Genetic algorithms are highly parallelizable, robust, semi-optimization algorithms of polynomial algorithm time complexity (26). These algorithms have been applied to a wide range of optimization problems and constitute a significant area of parallel algorithms research at AFIT (52, 6, 15, 73), and the scientific community at large.

10.2.1 Genetic Algorithm Research at AFIT. A genetic algorithm tool-box has been developed at AFIT to explore the capabilities of genetic algorithms. The tool-box has a sequential version implemented on the Sun 4 workstation, and a parallel version implemented on the Intel iPSC/2, iPSC/860 Hypercubes and the Paragon. To date function optimization applications using both standard and messy (52) genetic algorithms have been parallelized (15, 73). This work is based on the significant achievements by many researchers (26, 25, 66).

10.3 Overview of Genetic Algorithm Characteristics

The purpose of the genetic algorithm is similar to that of the mission routing A^* algorithm. The GA approach is simply a probablistic approach to finding an optimal solution to an optimization problem. Whereas the A^* algorithm uses a deterministic search technique to determine a solution, the GA algorithms uses a guided random search technique. In a genetic algorithm solution technique, string representations of complete or partial solutions are manipulated based on the mechanics of natural selection and natural genetics, the aim being to optimize an objective function. The objective function models the problem to be optimized. This function is defined in terms of the solution strings and can be used to evaluate the relative quality of solution strings.

The genetic algorithm searches the solution space by evolving a set of solutions. This initial set of solutions is generated randomly or based on domain specific knowledge, and is referred to as the initial population. Each new population generated is evolved using three basic operators; mutation, crossover, and reproduction (26). Thus the GA search process involves generation of new populations based on the current population. Each new population is referred to as a generation. Solutions are reproduced in later generations based on their quality as evaluated by the objective function to be optimized. The basic step of the genetic algorithm search process are (26:63).

1. Create initial population.
2. Create new population by applying:
    - Selection (Based on fitness).
    - Mutation (Based on defined probability).
    - Crossover (Based on a defined probability).
3. Repeat 2 until termination criteria is satisfied.

The termination criteria is completely flexible and may be based on time, number of generations, a variance of solution fitness, a solution quality figure, and many others.

10.4 Parallel Implementation of Genetic Algorithms

The basic genetic algorithm and its many variations are ideally suited to parallel implementation. Parallel implementations vary in the method used to distribute populations and evaluate generations. The most common parallel implementations involve the:

- Island model,
- Cellular model, or
- Deme model (globalized).

In the island model each processor is given a separate population, and evaluation of fitness, selection, mutation, and crossover are all performed on the local populations. In the deme, or globalized model, the schema on a number of processors are considered to form a population and in the extreme form, a single population. The distribution of an individual population on a cluster of processors is often referred to as the cellular model (52). In the global model the selection, evaluation, and then genetic operators are applied to distributed populations. The inclusion of schema migration probabilities allows further flexibility in the implementation. Schema migration allow strings to move between populations based on user specified criteria. This promotes sharing of genetic material between populations and can be implemented in both the basic approaches.

10.5 Application of Standard Parallel Visualization Tools

The insight provided by standard application independent visualizations is greatly dependent on the parallel implementation used. If the island model is used without migration, then individual processors do not communicate and the displays depict little interesting information. This results directly from the generic animations being based entirely on message traffic information. Once global evaluation, or schema migration is included, then resulting inter-node communications provide events that produce meaningful animations. The following sections outline characteristics

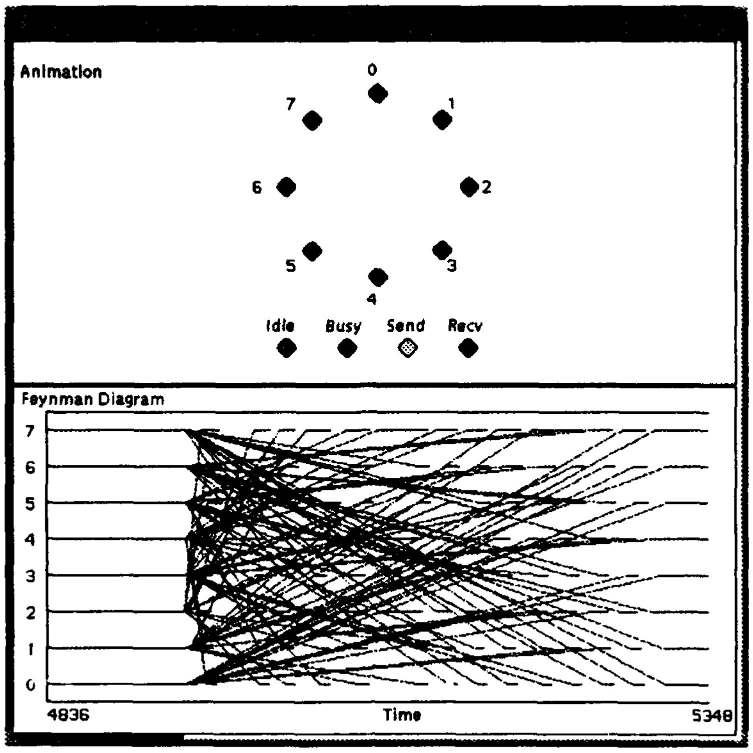


Figure 44. Schema Distribution for PSGA

of genetic algorithms that can be analyzed using standard visualizations included in the AAARF system (84).

10.6 Task Management

To effectively and efficiently map the GA to the target system, the combined effect of a number of characteristics must be considered. This can be generalized as task management and deals with partitioning, load balancing, and task scheduling.

10.6.1 Communications. If a global or cellular implementation model is used, then communication is required to perform the selection process. If schema migration is used, then the communication patterns generated by the distribution of solution strings can also be analyzed. The Feynman and Communication load displays (84), can be used to examine the effects of this communication. In particular, examination of the mapping to the processor architecture and the resultant message channel delays would be of interest.

The parallel genetic algorithm programs used at AFIT currently only include island model implementations, with schema migration (PSGA) (52). The Feynman diagram shown in Figure 44 highlights the importance of applying visualization to determine the effect of communication patterns. Figure 44 shows the effect of using the broadcast communication primitives to distribute schema between processors. A simpler ring based communication pattern would be the obvious alternative to this expensive original approach. The contention for communication channels significantly extends the transfer time. The displays can also be utilized to effectively analyze the global selection and evaluation process. Note that this animation is unable to communicate anything about the relative quality of migrating schema since basic animation events cannot record the data structure that constituted the message. Since this level of instrumentation cannot decode message traffic the quality, number, and characteristics of the shared schema is hidden from the user.

10.6.2 Examination of Load Balancing. In the current implementation of PSGA the activities of the individual processors are synchronized. This synchronization occurs after every so many generations with schema being shared between processors. However, this is not the only possible implementation and totally asynchronous implementations are used. With implementations that allow variable length schema and various population sizes, load imbalance can occur. This can be compounded if a centralized controller is used to distribute schema and populations dynamically. Under these situations the Utilization display (85) can depict this imbalance. The addition to the status panel of current generation information for each processor, permits global progress to be examined on a gross scale.

10.6.3 Examination of Computational Phases. The use of the computational phases display can assist in determining program profile information. In the case of the genetic algorithm, the regular nature of the computation phases provides simple analysis. The phase/task timing display (9) allows the computation cost of each phase to be examined.

10.7 A Need for Application Specific Displays

While standard displays provide some insight into the execution of genetic algorithms, much of the important information is obscured from the user. Information relating to program progress, solution quality, and other criteria are contained in the data structures manipulated by the genetic algorithm. To expose this information, we must analyze and display these data structures. In the case of the genetic algorithm, we are concerned with the types of schema in the distributed populations and general characteristics of these populations. This information can only be captured by implementing interesting events that record these data structures during program execution. The displays developed are also applicable to other evolutionary algorithms since the data collection system records fundamental program characteristics common to many evolutionary algorithms, in particular, the results of string evaluations, population size, and schema sharing between populations. The instrumentation approach taken assumes only minimal application program capabilities, basing most animations only on string fitness evaluations. This approach allows for implementations that don't evaluate other population parameters.

10.8 Analysis of Important Genetic Algorithm Trace Information

Control structure event tracing is provided by the standard PRASE (42) instrumentation suite. To provide execution tracing for data structure information, Interesting Event (IE) markers are included in the target program. We can consider the state of the genetic algorithm to be defined by the characteristic of the current generation (26). For each generation of a single population the important statistics are:

- Average population fitness.

- Variance in population fitness.
- Max and Min population fitness.
- Number of Mutations.
- Number of crossovers.
- Number of a given schema type.
- Sum of population fitness.
- Population size.
- Number of schema types represented.
- Average schema length.

The variable length schema are important in rule based GA systems where string length is also adapted as part of the search process (54). When extended to a multiprocessor environment we produce a number of sub-populations. Dependent on the implementation model, schema may or may not be transported between sub-populations. In addition to the parameters in the above list, the values of these parameters across all populations are also valuable. Information about the migration of schema between these sub-populations also needs to be depicted. This allows the monitoring of interaction between sub-populations and the analysis of different parallel implementation models:

- Single population statistics for each population.
- Number of migrations.
- Characteristics of migrating schema.
- Combined statistics for all populations.

Based on the requirement to analyze these execution parameters the following section outlines the instrumentation requirements.

10.9 Data Collection Requirements

To provide graphical representations of the important characteristics listed in Section 10.8, a significant amount of data must be recorded. In addition, the variety of algorithm types likely to be instrumented, eliminates direct recording of values for such statistics as population diversity and other characteristics that are not normally calculated during execution. Since many evolutionary algorithms use variable population sizes and schema lengths, even recording of a single generation evaluation is difficult. The common denominator between all evolutionary algorithms would appear to be the evaluation of string quality. This particular calculation usually results in a single quality value for a particular solution string. By recording these evaluations we can determine a number of our important population statistics. These include the average, minimum, maximum and variance in population fitness, both locally and globally. The number of events also indicates the population size. By recording events indicating the fitness of schema shared between populations, we provide migration information. The migration information includes, quality and volume information.

Information about the fitness of population schema only indicates the overall characteristics of the schema. This information reveals very little about the actual form of the solution strings. To reveal these features actual parameter values of the schema must be collected. Since solution schema can contain many parameter values, this resolution can be prohibitive however, it is necessary for a number of analysis problems.

10.9.1 Trace Volume. Realistic problems examined using genetic algorithms typically result in long program execution times. A typical execution of the protein folding energy minimization problem (6) could potentially consume hundreds of hours on a parallel machine. Since the creation of each generation represents a distinct point in the execution, we can consider the program to be simply a series of generations. The uniformity of most implementations means that each individual phase is representative of the next execution cycle. Thus, once the communications and utilization have been examined for a particular phase we need only consider the contents of the different populations. During long executions of the genetic algorithm, the contents of the population at a particular point in time are of limited value. Our main concern are the trends in the composition of these populations. In this manner, we need only ever examine generations as frequently as required, and in this way limit the tracing. For example, data from one generation per 200 produced, may provide enough detail in executions that contain 100000 generations.

10.10 Implementation Strategy

The visualizations developed provide relevant data without imposing a significant overhead on the program under examination. The final implementation allows flexibility in monitoring levels from a single global best solution value, to an extensive list of population parameters. The animations implemented here are based on the adaptive displays environment that was developed as part of this research, outlined in Chapter IX. Unlike the animations produced for mission routing and parallel discrete event simulations, these animations were produced without coding. This is the major advantage of the adaptive displays environment. To produce these animations, a display definition and event mapping file are developed. This file is then used by AAARF to create the specified animation.

The following steps were utilized in generating animations within the adaptive display generation environment.

1. *Determination of execution parameters required to be animated.* This step involves determining animation objectives and specification of display requirements.
2. *Event data required.* Based on the parameters to be examined determine which event types are required to produce this data.
3. *Selection of data transforms.* Select the predefined operations, listed in Section 9.2.5, to specify the data transforms required to translate event data into the required execution pa-

rameters. The requirements for event data scaling, linear/logarithmic and offset values are also determined.

4. *Display format.* Based on animation requirements determine display format. In particular examine the use of scrolling displays (windowed time interval) or snapshots (instantaneous value animations). Also specify update methods, from Section 9.2.3. Updates can be based on arrival of events or the passage of time.

5. *Animation Objects.* From the set of predefined graphical objects (Section 9.2.2) select specific instances to compose the required display. Define the initial conditions for object parameters and store in *display format file*, Appendix B. Assign specific animation parameters to graphical object parameters. Involves assigning the operations selected at Step 3.

6. *Event mapping.* Based on the input event set, create a mapping of *node & event* pairs to effect numbers (see Section 9.2.4). Produce a mapping of event to object parameters that require input data from this event. These maps are created using the format shown in Appendix B. They are stored with the object definitions in the *display format file.*

10.11 Data Collection for AAARF

Since this instrumentation was aimed at providing data input for the adaptive display environment, the pre-formatted IE events are used. The important fields in this event are the *event_type* and *data_value.* The *command* field is discussed in Section 9.2.1 and used to produce composite events, a feature not used in this application. The event type is thus formatted as follows:

$$IE((int)\ EVENT\_TYPE,\ (boolean)\ COMMAND,\ (float)\ data\_value)$$

Based on this basic event type the following instrumentation was inserted into the GA program. All data collection events are included in *pgav2.c.* A current instrumented implementation is available for both the iPSC/2 and iPSC/860, discussed in the AAARF Users' Guide (58). For this particular application, a specifically tailored version of the PRASE data program is utilized. This version, stored and compiled with the main PGA C files, is modified to record execution time in milli seconds not micro seconds.

To provide maximum flexibility, it has been assumed that the GA program only provides an evaluation of string fitness. From these events basic population parameters are determined, for example variance, population size, etc. Interesting event *NEW_GEN (int 99)* signifies that a new generation is about to be evaluated.

$$IE(NEW\_GEN, 1, NULL);$$

Interesting event *FITNESS (int 100)* is the base event value used to record the result of string evaluations. The first evaluation result *(float)performance* is recorded as event 100 and each following evaluation is recorded as the next consecutive event number. The evaluation of the first string from the next generation resets the event to (int 100).

$$IE(FITNESS + evaluation\_count,\ 1,\ (float)\ performance);$$

To record the migrating schema the following event types are inserted. The *BEST_POSITION (int 299)* event indicates the population position number of the migrating schema. The *MIGRATE (int 300)* event records the fitness value of the migrating schema. If more than one migration is performed per generation, the MIGRATE event marker is incremented in the same manner as the FITNESS event.

$$?EST\_POSTION,\ 1,\ (float)\ (Best\_guy));$$

$$IE(MIGRATE + migrate\_count,\ 1,\ (float)\ (New[Best\_guy].Perf));$$

Instrumentation to record the actual schema requires significantly more data storage. However, when examining true population diversity and composition, this data is essential. The genetic algorithm problem utilized for this study, protein folding (6), uses the binary string of the GA to represent bond angles. As with most GA programs, the binary representation is decoded to produce a vector of data values. These can be floating point numbers or integers. The instrumentation system simply records the individual vector components as numbered events. In this way the instrumentation is independent of string length. The following event type is utilized.

$$IE(SCHEMA\_VAL + string\_postion,\ 1,\ (float)\ value);$$

This method is also applicable to recording directly the binary strings if required. The SCHEMA_VAL event is defined as (int 500).

The PGA program utilized has the capability to calculate at run time, a number of population statistics. These can be instrumented if required, but tend to make the animations too program dependent. The standard set present here ensure that the instrumentation is compatible with a range of application programs.

10.12 Application Specific AAARF Animation Displays

The following section outlines the individual displays developed and discusses their capabilities. Each animation is specified by a display definition file. These files are contained in Appendix C. The flexibility of the adaptable displays environment promotes unique variations of the basic formats to meet specific animation objects. To use a particular animation, the relevant display definition file name is simply entered into the control panel input field.

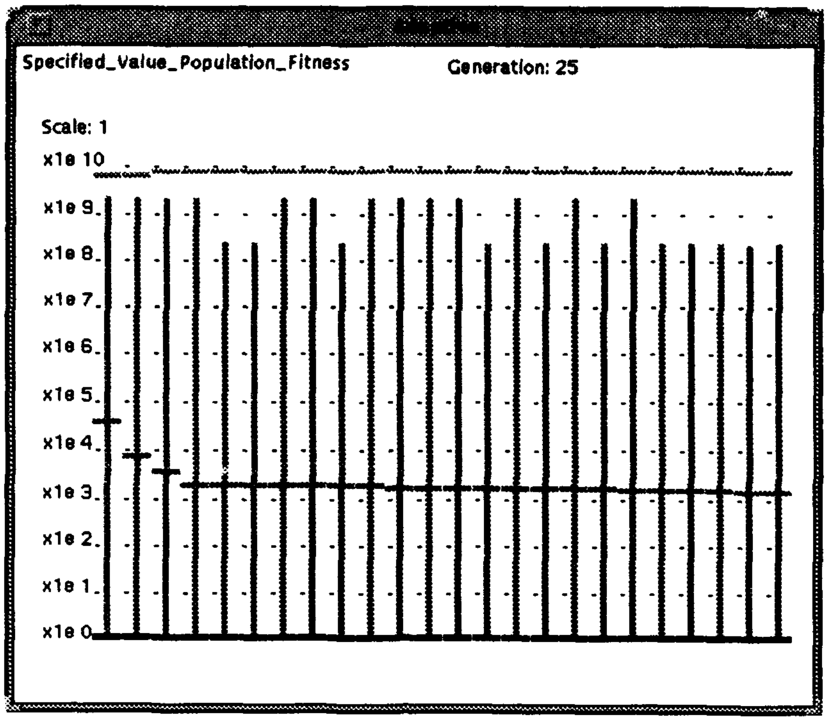


Figure 45. Scrolling Specified Value Display (Minimum, Maximum, and Average Fitness)

10.12.1 Specified Value Display. The specified value display allows the user to depict on both a static and scrolling bar graph, any interesting population parameters. For the scrolling animation, the horizontal time scale is measured in generations (see Figure 45). The static bar graph display shows the value of the specified parameter relative to each processing node. In this case the processors form the horizontal axis (see Figure 46).

It is possible to instantiate as many of these displays as required, with each display depicting any parameter of interest for any processor. Most displays allow more than one parameter to be displayed at the same time. The following list contains the parameter available.

- Average Fitness.
- Variance in Fitness.
- Max/Min Fitness.
- Number of Mutations.

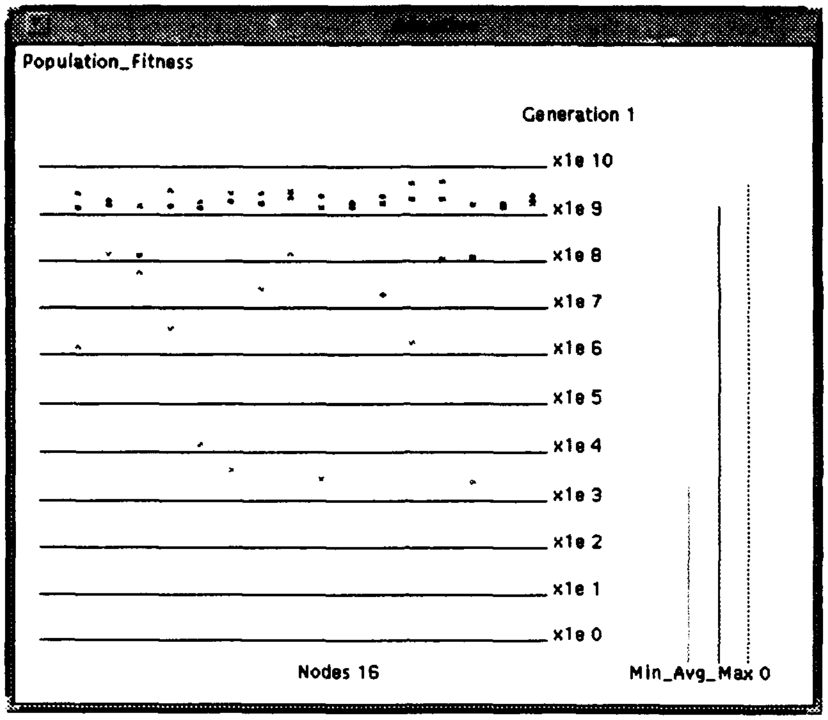
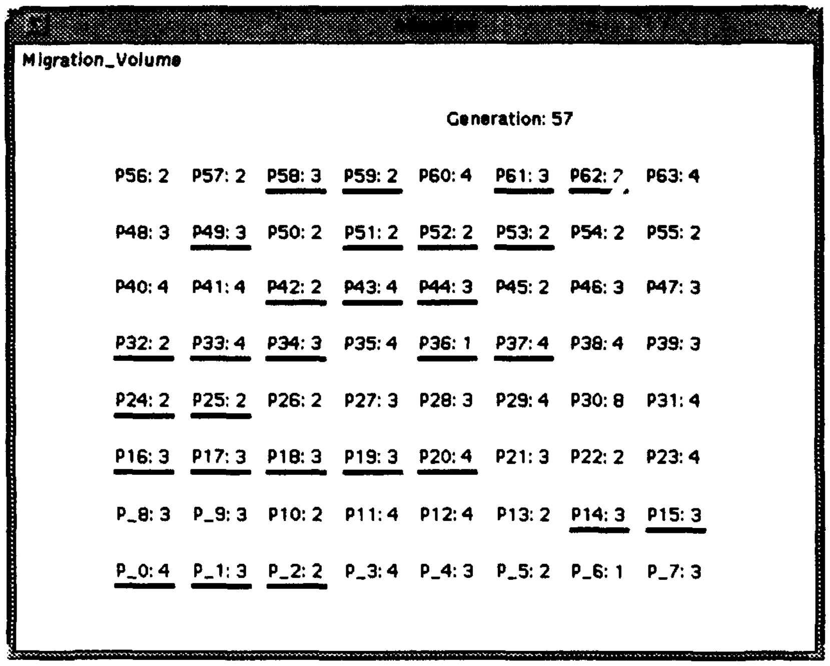


Figure 46. Static Specified Value Display (Minimum, Maximum, and Average Fitness)

- Number of Crossovers.
- Population size.
- Number of migrations.

10.12.2 Parameter Matrix Display. The matrix representation provides a depiction of interesting parameters relative to the processing elements. In this animation, the individual processors are represented and the current value of a particular parameter of interest is displayed. It is actually not necessary to depict all processors. By modifying the event map, statistics from multiple processors can be combined to produce an aggregate value. The parameters depicted include those listed in Section 10.12.1. The display includes color coding and bar graph representations that allow multiple parameters to be examined simultaneously. This can be particularly useful to depict variances and average values simultaneously. The displays can effectively depict over 1000 processors. The display is similar in format to the one shown in Figure 47 and includes an addi-



```
Migration_Volume

                              Generation: 57

P56: 2   P57: 2   P58: 3   P59: 2   P60: 4   P61: 3   P62: 7   P63: 4

P48: 3   P49: 3   P50: 2   P51: 2   P52: 2   P53: 2   P54: 2   P55: 2

P40: 4   P41: 4   P42: 2   P43: 4   P44: 3   P45: 2   P46: 3   P47: 3

P32: 2   P33: 4   P34: 3   P35: 4   P36: 1   P37: 4   P38: 4   P39: 3

P24: 2   P25: 2   P26: 2   P27: 3   P28: 3   P29: 4   P30: 8   P31: 4

P16: 3   P17: 3   P18: 3   P19: 3   P20: 4   P21: 3   P22: 2   P23: 4

P_8: 3   P_9: 3   P10: 2   P11: 4   P12: 4   P13: 2   P14: 3   P15: 3

P_0: 4   P_1: 3   P_2: 2   P_3: 4   P_4: 3   P_5: 2   P_6: 1   P_7: 3
```

Figure 47. Migration Display by Processor (Total and Last Distributions)

tional bar above the processor identification number to indicate the additional parameter. Once again, an unlimited number of variations using the same basic displays definition file are possible.

10.12.3 Migration Display. This animation is an adaption of the matrix display that depicts the sharing of schema between populations. In this animation the volume, fitness, and other statistics related to migrating schema are shown. The matrix display represents the interactions between population. In a cellular or global population model, the animation can depict the interactions between sections of the distributed population. An example of this display is shown in Figure 47. In this particular animation the values on the nodes represent the number of schema that have been distributed by that node, or cluster of nodes, during execution. The presence of a bar under the processor indicates that on the last sharing opportunity, this node distributed at least on schema. The GA program was configured in this case to distribute one schema every ten generations if the schema was better than any sent out by that node so far.

10.13 User Interaction

Genetic algorithms are well suited to tackling problems of large dimensionality. For these problems, computation time can typically increase exponentially. Even on massively parallel systems only a small fraction of the total search space can be examined in a reasonable time period. For current AFIT research into protein folding (6) this can easily amount to hundreds of hours of system time. The animations developed to date have focused on simply depicting the execution of the GA program. The real-time connection, possible with AAARF, could permit user interaction with the algorithm. By analyzing parameters such as solution variance, average fitness and population distribution, the user can determine how the search is progressing. This analysis could highlight populations that have converged on solutions or particular schema segments that appear to fit problem domain criteria. For example, in the protein folding problem, sections of schema could reflect meaningful properties in the solution space confirming particular sections of the final solution. Based on this information the user could actually modify program parameters during execution to produce superior results. The following sections outline this approach in more detail.

10.13.1 Closing the Loop. Once characteristics of the GA program's execution have been determined, the user could then inject into the executing program additional changes. This could be as simple as increasing the mutation rate to increase diversity in populations that have converged. Totally new populations could be generated to replace ones that are not producing promising solutions. Reductions in the mutation rate could be initiated to examine a local area in the search space more thoroughly. Solutions could be examined to determine important components of promising solutions and this genetic material selectively distributed across populations.

The system could also be configured to monitor particular aspects of the executing GA automatically. This could then be used to alert the operator to initiate changes. Experimentation with this higher level interaction could lead to the generation of rules that could automatically adapt the search process. On truly long executions, it is important to ensure maximum program performance. This constant monitoring and adaption can be promoted by the integration of real time visualization systems.

10.13.2 User Interface. The user interaction display would ideally be suited to implementations based on the matrix display, the base configuration depicting the variance and average value of population fitness. Attached to each processor display would be a menu containing the current GA parameters applicable to the population on that node. This would include:

- Mutation probability.
- Crossover rate.
- Migration criteria.

In addition to allowing the user to examine these values, they could be adapted during execution. This could be further extended to allow more advanced manipulation of the executing algorithm. In particular, destruction or creation of completely new populations would be possible.

10.14 Summary

Results on early use indicate that the animations provide valuable insight into the execution of parallel genetic algorithms. The number of interesting characteristics for more advanced GA implementations provides a limitless range of display formats. The AAARF system has proven that valuable formats can be generated relatively easily using the adaptable animation builder discussed in Chapter IX. Without this approach, significant coding would have been required for each application specific display. All these displays were developed without a single line of additional code. While the prototype implementation of the adaptive animation environment provides the basic functionality, a suitable user interface would greatly enhance the process of creating the display format file. This has not however, limited the capability of the resultant animation displays.

The possibility of user input into the executing GA, based on information gained from visualizations, could provide a focus of further research activity. This particular aspect could be generalized to cover other algorithms, particularly intensive searches demanded by optimization problems. User interaction is of particular interest since it is not currently included in the major animation systems.

XI. Conclusions and Recommendations

This chapter presents conclusions regarding the effectiveness of the enhancements made to AAARF as a result of this research. Recommendations for future modifications to AAARF and the development of a new AAARF implementation are also presented.

11.1 Conclusions

This research has focused on a number of elements vital to parallel algorithm animation. The primary research objective of providing improved animation support for AFIT research projects has been met. This has been achieved by integrating additional animations into the AAARF system and by development of an adaptive display building environment. Detailed analysis of the limitations inherent in current animations systems revealed significant factors that influence animation system design. These factors, listed below, have not been successfully addressed by other systems.

- *Animation of system level events is too restrictive.* This is due to limitations of trace volume and the resulting program perturbations.
- *Animations of algorithm data structures are essential.* Animations based only on communication events do not effectively depict many desired aspects of algorithm execution.
- *The capability to depict significant periods of program execution is required.* Many algorithm modifications effect execution parameters only when observed over significant periods of program execution.
- *Animation of abstract program data requires flexible display formats.* The animation system must be capable of providing animations that relate directly to user identifiable behavior. This requirement leads to user composible display formats.
- *Utilization is not simply processor idle time based on message passing.* Parallel algorithm implementations can often require additional processing not included in serial implementa-

tions, for example dynamic load balancing. These types of overheads directly effect system utilization statistics and must be included.

These factors lead to the design and development of enhancements to the AAARF system, listed below. The enhancements represent additional animation classes that directly address the limitations listed previously. The development is compatible with the object based implementation of AAARF and the user interface consistent with previous capabilities. The animations were extensively tested using current versions of AFIT research applications. This testing included student usage of the new animations for laboratory experimentation. In addition, results gained from program analysis using the new AAARF animations have already lead to enhancement of the target applications.

- Application specific displays for parallel discrete event simulations, multi-criteria mission routing A* search, and genetic algorithms.
- Improvements to general system level animations.
- The development of a trace file statistical analysis tool.
- Implementation of an adaptive system for building application specific animations has been implemented.

From a usability perspective, the AAARF system was widely used by students not associated with its development. While the majority of the instrumentation work was done by the author, software from six other students was instrumented with success.

11.1.1 Trace Data Collection and its Impact on Animation Systems. A detailed analysis of the effects of trace data collection was completed including a quantification of the PRASE data collection system overhead. This investigation revealed that the current system level animations. are ineffective when applied to the current generation of massively parallel computer systems. This problem is a direct result of the trace data volume required to generate system level animations.

This effectively limits this level of observation to examining very brief periods of execution. It can be concluded that trace volume and the ever increasing size of parallel processing systems require that animation systems' data requirements need to be reduced by many orders of magnitude. Simply providing scalable displays is of little benefit, since the trace data volume problem remains. Thus, visualization should focus research on animating higher level algorithm information. A model for future animation system development at AFIT is proposed, based on this principle. The model includes the following important characteristics:

- Architecturally independent self defining event tracing.
- Top down trace resolution.
- User defined and configurable event filtering.
- Integrated multidimensional data visualizer.
- User specified interactive graphical display builder.
- Pseudo real time display presentation.
- Pseudo real time user manipulation of algorithm.
  - event tracing level.
  - data structure values.
- Animations built on higher performance graphical workstations (SGI etc).

Trace data standardization is still an open question for the scientific community at large. However, Intel's adoption of the Pablo (62) self defining event tracing provides a focus for continued AFIT research. AFIT's long association with the Intel product line is continuing with the Intel Paragon system in the near future. The adoption of the Pablo event tracing as a standard shall not restrict research on other systems such as the CM-5 (81) since it is currently available for this and many other system types.

11.1.2 The Adaptive Display Generation Environment. The most promising result from this research is the effective use of adaptive application specific display. When applied at the appropriate level, they provided greater insight than generic animations. The ability to define display formats for a particular application without resorting to coding is a significant improvement over other systems. The generalized mapping of events to any parameter of a graphical object provided virtually unlimited scope for animation generation. This adaptability not only ensures that the system can provide the required animations, but also enhances machine and algorithm independence. The utility of this approach was clearly demonstrated with the development of displays for parallel genetic algorithms. The simplification of event types also appears particularly beneficial with the elimination of instrumentation complexities introduced by more complex structures.

The ability to produce arbitrary animations relatively simply promotes both the education and program development capabilities of any parallel algorithm visualization system, such as AAARF. The flexibility in display formats addresses the problem of producing useful analysis animation for program development. In addition, the event mapping process effectively deals with defining displays for massively parallel systems. This flexibility also introduces the opportunity to produce animations that demonstrate specific algorithm behavior. By allowing displays that combine depictions of algorithm data and system level behavior, superior instructional animation can be produced. This, combined with a realistic real-time data requirement, provides an effective system for support of parallel program development.

The actual software developed for the adaptive display environment was designed to be independent of the AAARF system. The additional modules included into AAARF require only trivial data from the remainder of the system. This requirement is limited to event data, the current animation time, and display canvas pointers. The modular design of these additions are ideally suited for inclusion in other animation systems.

11.2 Recommendations

The following recommendations are made based on the results of this research and a comprehensive review of other state-of-the-art parallel animation systems. In defining these recommendations, the guiding principles have been:

1. Providing future search activity that ensures AFIT's algorithm research is relevant to the general scientific community.
2. Research in this area should leverage the technology developed with the AAARF system, other research activities, and the best commercially available products.
3. The system of choice is the Intel product line, and in particular, the Paragon (39) system. However, portability is vital to allow comparison with execution on other systems such as the CM-5 (81).

The recommendations can be divided into three distinct areas and the following discussion reflects this organization.

11.2.1 The Current AAARF System. The following recommendations relate directly to the current AAARF implementation.

1. ***Further development on the current system software should be discontinued.*** The system is too restrictive for serious future development activity for massively parallel computers.
2. ***AAARF should continue to be used for animation of the algorithm classes for which application specific displays have been developed.*** These displays are not readily found in other systems and the development effort required to regenerate them would be substantial. These animations provide tremendous insight into the execution of parallel programs and are ideal for educational objectives.

3. ***AFIT should adopt the Intel ParAide (27) development and visualization environment for system level animation activities.*** Vendor support and development would ensure that other AFIT researchers are using the most reliable animation system available. Intel's goal of including the complete Pablo (62) animation environment, a leading edge research project at the University of Illinois, should ensure that it is also the most capable system. The results obtained by AFIT researchers on this system would then be compatible with systems used by other researchers. The most compelling reason for this recommendation is that this level of animation, based on this research, appears unable to address the scalability problems of massively parallel systems.

4. ***The serial animation component of AAARF should become a separate application program.*** This component of AAARF is relevant irrespective of future developments. It will remain an important teaching aid for AFIT algorithm based classes.

11.2.2 The Next Generation of AAARF. A new animation tool must be developed based on the outline provided in Chapter VIII. This will effectively constitute a second generation of the AAARF system, not an incremental improvement. Only these steps will ensure that AFIT parallel algorithm animation remains relevant in the future.

1. ***The new system should be developed as an extension of the Intel ParAide tool (27)*** . This will ensure maximum leverage of capabilities from the current generation of systems. One of the most disappointing aspects of AAARF has been its inability to utilize the capabilities of other systems, such as display formats. Currently ParAide does not support application specific animations. The capabilities of the adaptable animation environment, discussed in Chapter IX, would significantly enhance the ParAide. The modular design of ParAide will allow development without significant interaction with Intel.

2. ***The system should utilize the Pablo event tracing technique.*** The reasons for this are clearly discussed in section (55). The most effective approach here would be to define an equivalent to

the AAARF IE() event type. The work on adaptive display for application specific animations would then be directly applicable.

3. *Low level event tracing should not be considered part of future AFIT animation research.* This section of the animation task should be left to the ParAide (62) system. Future work in this area will increasingly require operation system modification and joint development with vendors, a difficult path for AFIT to pursue. From this research it would also appear that this level of observation is not applicable to the majority of AFIT's animation requirements.

4. *Research should focus on the development of an interactive animation generation system for high level algorithm displays.* User configurability and menu based display format generation provides the most promising field of future AFIT research. This was clearly demonstrated by the adaptable animation environment developed as part of this research (see Chapter IX). Higher level algorithm animation provides the only way of obtaining the orders of magnitude reduction in trace data volume.

5. *Research activity should be moved to a High Speed Graphics System.* The display capabilities of the Sun workstation significantly limited the animation speed possible in the current AAARF system. To ensure that this problem is minimized in the next system, other graphics platforms, such as the Silicon Graphics systems, should be examined.

11.3 Impact on General Parallel Research at AFIT

Architectural independence for parallel software is vital if previously developed software is to be reused. Much could be gained by utilizing one of the many portable communication libraries, ie PICL (23). While a small overhead penalty is incurred, this is more than compensated for by the ability to obtain results on other architecture and eliminate version proliferation of research software. This type of decision obviously impacts the development of animation tools.

Appendix A. Data Collection Trace Formats

A.1 Trace File formats

The specification of the trace file format greatly influences the overall animation system. This appendix provides a brief overview of the major trace formats of interest to this research; PRASE (42), Pablo (62), and PICL (23). These three systems all represent different approaches to trace file format.

A.2 The PRASE Trace File Format

PRASE uses two trace files to record execution data. The first records system level events and program trace markers consisting of two to three data values (trace.dat). The second trace file is used for storage of large data vectors (algtrace.dat). The more important file is the *trace.dat* file, which is the only one that provides event buffering and thus can be used for accurate event tracing. Without at least limited event buffering, execution traces become meaningless.

The PRASE *trace.dat* file uses a fixed size event record regardless of what data actually is to be recorded. This allows for ease of implementation but significantly increases trace file size. The format was originally chosen for compatibility reasons which have long since passed (42). The PRASE event record format is shown below:

```
typedef struct
(
 char     record'type[6];
 long     recording'node;
 long     recording'pid;
 unsigned long begin'time;
 unsigned long end'time;
 union
 (
  struct
  (
    long channel'num;
    long message'type;
    long message'size;
    long message'count;
    long addressed'node;
    long addressed'pid;
    long position'marker;
  ) message;
  char   char'marker[CHAR'MARK'LENGTH];
  double double'marker;
  float  float'marker;
  int    int'marker;
  long   long'marker;
  short  short'marker;
 ) data;
) RECORD;
```

As can be clearly seen from the event typedef, even if a single integer value and time stamp are recorded, a significant amount of empty data fields are stored in the event trace. One further disadvantage of the PRASE trace file format is that it is stored in binary format. While this reduces the total file size, it makes it impossible to examine using unix commands such as *grep*. As a result, an ASCII duplicate is often made to allow examination of the file.

A.2.1 Event Requirements for the Adaptive Display Environment. The work on adaptive display bases all its event requirements on a time stamped node number, integer, and float data tuple IX. While the current implementation uses the PRASE event records, the remaining data fields are not used. This is not the case for other AAARF animations. Thus it would be possible to reduce the PRASE record format if compatibility with previous research was not required. An example format would be:

```
typedef struct
 (
long     recording'node;
long     recording'pid;
unsigned long time;
long     event'type;
int      command'marker;
float    float'marker;
)RECORD;
```

Rather than use complex event types the *command_marker* allows events to be chained together. Currently all data values recorded by the animation system are converted to float type. While this approach does not allow text strings to be recorded, it could be added if required. The current adaptive display implementation allows the *event_types* to be mapped to text message objects and thus any textual messages can be generated in this manner.

A.3 The PICL Trace Data File Format

The PICL instrumentation system takes a similar approach to PRASE. All events are strongly typed with a specified format. however, the implementation does not require unused data fields. PICL uses an ASCII text file as the storage format and writes the data directly to this file. The data is encoded as one of a number of event types and occupies a single line in the trace file. The trace file is then read as a character string, line by line, and the first integer on each line represents the format number of that event. The advantage of this approach is that an event of irregular size can be recorded. However, the user is restricted to predefined event formats. The following is a representative section of a PICL trace file and its translated meaning:

```
4 0 489 47 15 30032 1
11 0 489 47 0 0
4 0 491 17 49 30032 1

** Which translates to:

send clock       at 0489 node 47 to node 15 type 30032 lth 1
compstats clock  at 0489 node 47 idle 0 0
```

```
send clock      at 0491 node 17 to node 49 type 30032 lth 1
```

While the text file format is less compact than a binary form, the use of variable sized event messages actually provides more compact results. The format is also particularly suitable for browsing and analysis outside the animation environment. The only real limitation here is that the user is restricted to predefined event formats.

A.4 The Pablo Trace Data Format

Pablo represents one of the most elaborate instrumentation system currently available. It allows the user to specify event formats and provides buffering, at the nodes, for these irregular sized events. The central component is the Self-Describing Data Format (SDDF) which is a trace description language that specifies both the structure of data records and data record instances. The format supports the definition of records that contain scalars and arrays of the base types found in most programming languages. In addition, SDDF supports the definition of multi-dimensional arrays whose sizes can differ in each recording instance. The Pablo system avoids the limitations associated with fixed event formats.

Pablo's SDDF meta-format also eliminates another problem by supporting both ASCII and binary versions of the trace file. Tools are provided to convert from one format to another and the visualization tools can accept either. The following is an example of an SDDF event definition. Note the similarities with a C programming language structure definition.

```
#113:
//"description" "Loop Entry Trace Record"
"Loop Entry Trace" (
// "Time" "Timestamp"
int "Timestamp"
// "Seconds" "Floating Point Timestamp"
double "Seconds";
// "ID" "Event ID"
int "Event Identifier";
// "Node" "Processor Number"
int "Processor Number";
// "Loop" "Loop Index"
int "Loop Index";
// "Byte" "Source Byte Offset"
int "Source Byte";
// "Line" "Source Line Number"
int "Source Line";
);
```

The Pablo system is supplied a range of predefined event types to aid users of the system with program instrumentation. It is obvious from the flexibility provided that virtually any trace requirement can be met with the Pablo system. It would be particularly beneficial in providing events for the adaptive display environment developed in Chapter IX. In this instance only the definition of a single event type would be required. It will be interesting to see if users of the system take full advantage of complex event definitions, due to the added effort involved. It is likely that the majority of users will make do with the pre-specified event types, even if they are inappropriate.

Appendix B. Display Definition File Format

The ***Display Format*** is used to specify a set of animation displa\ ɜ. They can contain definitions for up to four display formats. Each frame is specified using the same basic format which contains the following items:

- Frame Definition line.
- Node & Event to Effect mapping table.
- Effect to Object, Parameter, and Operation map.
- Object Definition (Including Scale Factor).

To construct a file, the user system inserts the appropriate mapping and other parameters into the template shown below. Fields such as the event map, effect map, and scale factors can contain variable numbers of entries and are terminated with a negative integer. The number of objects contained in a particular frame are also user definable and the system continues to read objects for the current frame until an invalid object is read. In this case, the object is specified to be invalid if a NULL valid is read for the object VALID parameter. After reading an invalid object, the system expects to find a new frame definition. This process stops when an invalid frame is read, specified in the same method used for objects. A detailed discussion of this format is contained in the AAARF User's Guide (58).

```
/* A Display Specification Note:
/* The position and line spacing in the file is important.
/************************** FRAME ONE DATA *************************/ Must Be on Line 3.
VALID UPDATE'STYLE SCROLL'WIDTH SAMPLE'PERIOD UPDATE'EFFECT CLEAR'DISPLAY DISPLAY'NAME
#### NODE  EVENT  EFFECT
NODE  EVENT'TYPE  EFFECT
NODE  EVENT'TYPE  EFFECT
NODE  EVENT'TYPE  EFFECT
-1 -1 -1
```

```
#### EFFECT  OBJECT  PARAMETER  OPERATION
EFFECT  OBJECT  PARAMETER OPERATION
EFFECT  OBJECT  PARAMETER OPERATION
EFFECT  OBJECT  PARAMETER OPERATION
-1 -1 -1
# Object 0 #
VALID  OBJECT'TYPE  POSITION'X1  POSITION'Y1  COLOR  UPDATED  ACTIVE
DOT'SIZE RECT'HEIGHT STRING'VALUE TEXT'STRING DEGREES POSITION'X2 POSITION'Y2 RECT'WIDTH
 Scale Factors
PARAMETER   SCALE'FACTOR OFFSET LOG'SCALING
-1 -1
# Object 0 #
VALID  OBJECT'TYPE  POSITION'X1  POSITION'Y1  COLOR  UPDATED  ACTIVE
DOT'SIZE RECT'HEIGHT STRING'VALUE TEXT'STRING DEGREES POSITION'X2 POSITION'Y2 RECT'WIDTH
 Scale Factors
PARAMETER   SCALE'FACTOR OFFSET LOG'SCALING
-1 -1
# Object #
0
/************************** FRAME TWO DATA *************************/ .
VALID UPDATE'STYLE SCROLL'WIDTH SAMPLE'PERIOD UPDATE'EFFECT CLEAR'DISPLAY DISPLAY'NAME
#### NODE  EVENT  EFFECT
NODE  EVENT'TYPE  EFFECT
NODE  EVENT'TYPE  EFFECT
NODE  EVENT'TYPE  EFFECT
-1 -1 -1
#### EFFECT  OBJECT  PARAMETER  OPERATION
EFFECT  OBJECT  PARAMETER OPERATION
EFFECT  OBJECT  PARAMETER OPERATION
EFFECT  OBJECT  PARAMETER OPERATION
-1 -1 -1
# Object 0 #
VALID  OBJECT'TYPE  POSITION'X1  POSITION'Y1  COLOR  UPDATED  ACTIVE
DOT'SIZE RECT'HEIGHT STRING'VALUE TEXT'STRING DEGREES POSITION'X2 POSITION'Y2 RECT'WIDTH
 Scale Factors
PARAMETER   SCALE'FACTOR OFFSET LOG'SCALING
-1 -1
# Object 0 #
VALID  OBJECT'TYPE  POSITION'X1  POSITION'Y1  COLOR  UPDATED  ACTIVE
```

```
DOT'SIZE RECT'HEIGHT STRING'VALUE TEXT'STRING DEGREES POSITION'X2 POSITION'Y2 RECT'WIDTH
 Scale Factors
PARAMETER    SCALE'FACTOR OFFSET LOG'SCALING
-1 -1
# Object #
0
/************************** INVALID FRAME *************************/ .
0
```

Example specification files are contained in Appendix C. These files were used to generate some of the displays shown in Chapter X.

Appendix C. Definition Files for Adaptive Displays

C.1 Display Definition File for Process Timing Display (Figure 11)

A Display Specification Process Timing Event (99) Set Event and Node ID.

The position and line spacing in the file is important.

```
/************************** FRAME DATA *************************/

1 8 10 0 0 1 Phase`Execution`Time

#### NODE EVENT EFFECT

6 99 0

-1 -1 -1

#### EFFECT OBJECT PARAMETER OPERATION

0 0 9 8

0 1 7 8

0 2 9 3

-1 -1 -1

# Object 0 #

1 3 370 500 1 1 1

0 0 0 Interval: 0 0 0 0

 Scale Factors

9 1 0 0

-1 -1

# Object 1 #

1 2 492 1 70 1 1

0 0 0 null 0 0 0 7

 Scale Factors

7 2630 0 0

-1 -1

# Object 2 #

1 3 270 500 1 1 1

0 0 0 Occurence: 0 0 0 0

 Scale Factors

9 1 0 0

-1 -1

# Object 3 #

1 3 2 450 1 1 1

0 5 100 Scale: 0 0 0 19

 Scale Factors

-1 -1

# Object 4#
```

```
1 2 490 1 1 1 1
0 3 0 null 0 2 2 10
 Scale Factors
3 1 0 0
-1 -1
# Object ****************** LABELS ***************************#
1 3 3  38 1 1 1
0 0 2  + 0 0 0 0
 Scale Factors
-1 -1
# Object  #
1 3 3  76 1 1 1
0 0 4 + 0 0 0 0
 Scale Factors
-1 -1
# Object  #
1 3 3  115 1 1 1
0 0 6  + 0 0 0 0
 Scale Factors
-1 -1
# Object  #
1 3 3  153 1 1 1
0 0 8 + 0 0 0 0
 Scale Factors
-1 -1
# Object  #
1 3 3  192 1 1 1
0 0 10 + 0 0 0 0
 Scale Factors
-1 -1
# Object  #
1 3 3  230 1 1 1
0 0 12  + 0 0 0 0
 Scale Factors
-1 -340
# Object  #
1 3 3  270 1 1 1
0 0 14 + 0 0 0 0
 Scale Factors
```

```
-1 -1
# Object  #
1 3 3  308 1 1 1
0 0 16 + 0 0 0 0
 Scale Factors
-1 -1
# Object  #
1 3 3  346 1 1 1
0 0 18 + 0 0 0 0
 Scale Factors
-1 -1
# Object  #
1 3 3  384 1 1 1
0 0 20 + 0 0 0 0
 Scale Factors
-1 -1
# Object  #
1 3 3  423 1 1 1
0 0 22 + 0 0 0 0
 Scale Factors
-1 -1
# Object  ******************* LINES ****************************#
 1 5 492 38 1 1 1
0 0 0 null 0  494 38 0
 Scale Factors
-1 -1
# Object  #
 1 5 492 76 1 1 1
0 0 0 null 0  494 76 0
 Scale Factors
-1 -1
# Object  #
 1 5 492 115 1 1 1
0 0 0 null 0  494 115 0
 Scale Factors
-1 -1
# Object  #
 1 5 492 153 1 1 1
0 0 0 null 0  494 153 0
```

```
 Scale Factors
-1 -1
# Object  #
 1 5 492 192 1 1 1
0 0 0 null 0  494 192 0
 Scale Factors
-1 -1
# Object  #
 1 5 492 230 1 1 1
0 0 0 null 0  494 230 0
 Scale Factors
-1 -1
# Object  #
 1 5 492 270 1 1 1
0 0 0 null 0  494 270 0
 Scale Factors
-1 -1
# Object  #
 1 5 492 308 1 1 1
0 0 0 null 0  494 308 0
 Scale Factors
-1 -1
# Object  #
 1 5 492 346 1 1 1
0 0 0 null 0  494 346 0
 Scale Factors
-1 -1
# Object  #
 1 5 492 384 1 1 1
0 0 0 null 0  494 384 0
 Scale Factors
-1 -1
# Object  #
 1 5 492 423 1 1 1
0 0 0 null 0  494 423 0
 Scale Factors
-1 -1
 # Object #
0
```

```
/************************** FRAME DATA *************************/

0
```

C.2 *Display Definition File for Scrolling Specified Value Display(Figure 45)*

```
A Display Specification Note: Event (99) Set Event and Node ID.

The position and line spacing in the file is important.

/************************** FRAME DATA *************************/

1 8 20 0 55 0 Specified`Value`Population`Fitness

#### NODE EVENT EFFECT

0 99 55

0 100 0

0 101 1

0 102 2

0 103 3

0 104 4

0 105 5

0 106 6

0 107 7

0 108 8

0 109 9

0 110 10

0 111 11

0 112 12

0 113 13

0 114 14

0 115 15

0 116 16

0 117 17

0 118 18

0 119 19

0 120 20

0 121 21

0 122 22

0 123 23

0 124 24

0 125 25

0 126 26

0 127 27

0 128 28

0 129 29
```

```
0 130 30
0 131 31
0 132 32
0 133 33
0 134 34
0 135 35
0 136 36
0 137 37
0 138 38
0 139 39
0 140 40
0 141 41
0 142 42
0 143 43
0 144 44
0 145 45
0 146 46
0 147 47
0 148 48
0 149 49
0 150 50
-1 -1 -1
#### EFFECT OBJECT PARAMETER OPERATION
0  0  3 1
1  1  3 1
2  2  3 1
3  3  3 1
4  4  3 1
5  5  3 1
6  6  3 1
7  7  3 1
8  8  3 1
9  9  3 1
10 10 3 1
11 11 3 1
12 12 3 1
13 13 3 1
14 14 3 1
15 15 3 1
```

```
6 6 3 1
7 7 3 1
8 8 3 1
9 9 3 1
10 10 3 1
# Object 1#
1 1 485 50 70 1 1
4 0 0 null 0 2 2 0
 Scale Factors
3 130 0 1
-1 -1
# Object  3 #
1 3 270 500 1 1 1
0 0 0 Generation: 0 0 0 0
 Scale Factors
9 1 0 0
-1 -1
# Object   #
1 2 480 36 1 1 1
0 5 0 null 0 0 0 19
 Scale Factors
-1 -1
# Object   #
1 3 2 450 1 1 1
0 5 1 Scale: 0 0 0 19
 Scale Factors
-1 -1
# Object  ******************* LINES ****************************#
 1 5 482 38 1 1 1
0 0 0 null 0  484 38 0
 Scale Factors
-1 -1
# Object  #
 1 5 482 76 1 1 1
0 0 0 null 0  484 76 0
 Scale Factors
-1 -1
# Object  #
 1 5 482 115 1 1 1
```

```
0 0 0 null 0  484 115 0
 Scale Factors
-1 -1
# Object  #
 1 5 482 153 1 1 1
0 0 0 null 0  484 153 0
 Scale Factors
-1 -1
# Object  #
 1 5 482 192 1 1 1
0 0 0 null 0  484 192 0
 Scale Factors
-1 -1
# Object  #
 1 5 482 230 1 1 1
0 0 0 null 0  484 230 0
 Scale Factors
-1 -1
# Object  #
 1 5 482 270 1 1 1
0 0 0 null 0  484 270 0
 Scale Factors
-1 -1
# Object  #
 1 5 482 308 1 1 1
0 0 0 null 0  484 308 0
 Scale Factors
-1 -1
# Object  #
 1 5 482 346 1 1 1
0 0 0 null 0  484 346 0
 Scale Factors
-1 -1
# Object  #
 1 5 482 384 1 1 1
0 0 0 null 0  484 384 0
 Scale Factors
-1 -1
# Object  #
```

```
1 5 482 423 1 1 1

0 0 0 null 0  484 423 0

 Scale Factors

-1 -1

# Object  ****************** LABELS ***************************#

1 3 3  38 1 1 1

0 0 0  + 0 0 0 0

 Scale Factors

-1 -1

# Object  #

1 3 3  76 1 1 1

0 0 2  + 0 0 0 0

 Scale Factors

-1 -1

# Object  #

1 3 3  115 1 1 1

0 0 4  + 0 0 0 0

 Scale Factors

-1 -1

# Object  #

1 3 3  153 1 1 1

0 0 6 + 0 0 0 0

 Scale Factors

-1 -1

# Object  #

1 3 3  192 1 1 1

0 0 8 + 0 0 0 0

 Scale Factors

-1 -1

# Object  #

1 3 3  230 1 1 1

0 0 10  + 0 0 0 0

 Scale Factors

-1 -340

# Object  #

1 3 3  270 1 1 1

0 0 12 + 0 0 0 0

 Scale Factors

-1 -1
```

```
# Object #
1 3 3 308 1 1 1
0 0 14 + 0 0 0 0
 Scale Factors
-1 -1
# Object #
1 3 3 346 1 1 1
0 0 16 + 0 0 0 0
 Scale Factors
-1 -1
# Object #
1 3 3 384 1 1 1
0 0 18 + 0 0 0 0
 Scale Factors
-1 -1
# Object #
1 3 3 423 1 1 1
0 0 20 + 0 0 0 0
 Scale Factors
-1 -1
 # Object #
0
/************************* FRAME DATA ************************/
0
```

Bibliography

1. Beard, Capt Ralph A. ***Determination of Algorithm Parallelism in NP-Complete Problems for Distributed Architectures***. MS thesis, AFIT/GCE/ENG/90M-1, Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, March 1990.
2. Bentley, Jon L and Brian W Kernighan. "A System for Algorithm Animation," *Computing Systems*, *4* (1):5–31 (December 1991).
3. Bode, Arnde. "TOPSYS : TOols for Parallel SYstems." *Proceedings of the 4th European IPSC Users Group Meeting*. Intel, April 1992.
4. Brassard, Giles and Paul Bratley. *Algorithmics: Theory and Practice* (First Edition). Englewood Cliffs NJ: Prentice Hall, 1988.
5. Breeden, Thomas A. *Parallel Simulation of Structural VHDL Circuits on Intel Hypercubes*. MS thesis, AFIT/GCE/ENG/92D-01, Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, December 1992.
6. Brinkman. *Must add*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.
7. Chandy, K.M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24* (11):198–206 (April 1981).
8. Chandy, K.M. and J. Misra. "Distributed Deadlock Detection." *ACM Transactions on Computer Systems*, *1* (2):144–156 (May 1983).
9. Chase, Paul W. *Application Specific Animations for Parallel Algorithms*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.
10. Chaudhary, Vipin and J. K. Aggarwal. "A Generalized Scheme for Mapping Parallel Algorithms," *IEEE Transactions on Parallel and Distributed Systems*, *4* (3) (March 1993).
11. Comerford. Richard. "Alpha Origin," *IEEE Spectrum*, 26 (July 1992).
12. Couch, Alva L and David W Krumme. "Portable Execution Traces for Parallel Program Debugging and Performance Visualization," *IEEE Computer*, 441–446 (October 1992).
13. Dongarra, J.J and others. *LINPACK User's Guide to*. SIAM Pub, 1979.
14. Droddy, Vincent A. *Multicriteria Mission Route Planning using Parallelized A* Search*. MS thesis, AFIT, December 1993.
15. Dymek, Capt Andrew. *An Examination of Hypercube Implementations of Genetic Algorithms*. MS thesis, AFIT/GCE/ENG/92-M, Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, March 1992 (DTIC Number - Not Yet Assigned).
16. Fife, Keith C. *Graphical Representation of Algorithmic Processes*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1989.
17. Fineman, Charles E and Philip J Hontalas. "Selective Monitoring Using performance Metric Predicates." *Proceedings of Scalable High Performance Computing Conference SHPCC-92*. 162 - 165. Piscataway, NJ: IEEE Press, 1992.
18. Francioni, Joan M, et al. "The Sounds of Parallel Programs." *Proceedings of the 6th Annual Distrubuted Memory Computing Conference.*. 562–591. Piscataway, NJ: IEEE Press, 1991.
19. Friedell, Mark, et al. "Visualizing the Behavior of Massively Parallel Programs." *Proceedings of the Supercomputing 91 Conference*. 472–481. ACM, November 1991.

20. Fujimoto, Richard M. "Parallel Discrete Event Simulation." *Proceedings of the 1989 Winter Simulation Conference*. 1–34. 1989.

21. Fujimoto, Richard M. *Parallel Discrete Event Simulation*. Technical Report, Georgia Institute of Technology, 1992.

22. Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, California: W. H. Freeman and Company, 1979.

23. Geist, G. A., et al. *PICL: A Portable Instrumented Communication Library*. Technical Report, Mathematical Sciences Section, Oak Ridge National Laboratory, 1992.

24. Goldberg, Aaron J and John L Hennessy. "Performance Debugging Shared Memory Multiprocessor Programs with MTOOL." *Proceedings of the Supercomputing 91 Conference*. 481–491. ACM, November 1991.

25. Goldberg, David E. *Optimal Initial Population Size for Binary-Coded Genetic Algorithms*. Technical Report, Tuscloosa AL: University of Alabama, 1985.

26. Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading MA: Addison-Wesley Publishing Company, 1989.

27. Guist, Judy. "ParAide SSD Development Tools," *Proceedings of the Intel Supercomputer Users Group* (November 1993).

28. Heath, Michael T. "Visual Animation of Parallel Algorithms for Matrix Computations." *Proceedings of the Fifth Distributed Memory Computing Conference*. 1990.

29. Heath, Michael T and Jennifer A Etheridge. "Visualizing the Performance of Parallel Programs," *IEEE Software*, 29–39 (September 1991).

30. Heller, Dan. *XView Programming Manual*. Sebastopol CA: O'Reilly & Associates, Inc, 1991.

31. Hennessy, John L. and David A. Patterson. *Computer Architecture: a Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.

32. Holland, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor MI: The University of Michigan Press, 1975.

33. Hotchkiss, Robert S and Cheryl L Wampler. "The Auditorialization of Scientific Information." *Proceedings of Supercomputing '91*. 453 – 461. 1991.

34. Hwang, Kai and Faye A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

35. II, James. J Grimm. *Solution of a Multicriteria Aircraft Routing Problem Utilzing Parallel Search Techniques*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

36. Intel, Corparation, editor. *Proceedings of the 1993 Annual Intel User's Conference*, 1993.

37. Intel, Corporation, "Paragon XP/S System Specifications." Supercomputing Division Release Information, 1993.

38. Intel Corporation, Beaverton, Oregon. *Interactive Parallel Debugger Manual* (1 Edition), April 1993.

39. Intel Corporation, Supercomputer Systems Divisision, 15201 N.W. Greebreier Parkway Beaverton, Oregon 97006. *PARAGON Systems Manual*, April 1993.

40. Jacobson, Peter and Erik Tarnvik. "A High Resolution Timer for the Intel iPSC/2 Hypercube." *Proceedings of the 4th European IPSC Users Group Meeting*. unnumbered. Intel, April 1991.

41. Janzow, Pete. *MATLAB users manual*. The MathWorks, Inc, 1992.

42. Kahl, Mark Albert. *PRASE: Instrumentation Software for the Intel iPSC Hypercube*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1988.

43. Lack, Michael D. and Gary B. Lamont. "Visualization of Hypercube Pedagogical Parallel Programs." *Proceedings of the Hawaii International Conference on System Sciences (Software Technology)2*. IEEE Computer society Press, 1992.

44. Lack, Michael D. A. *Enhanced Graphical Representation of Parallel Algorithmic Processes*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

45. Lamont, Gary B. and et al. *AFIT Compendium of Parallel Programs* (3rd Edition). Air Force Institute of Technology, WPAFB, OH, 1992.

46. Lewis, Ted G. and Hesham El-Rewini. *Introduction to Parallel Computing*. Englewood Cliffs, NJ: Prentice Hall, 1992.

47. Lo, Virgina M, et al. "METRICS: A Tool for the Display and Analysis of Mappings in Message-passing Multicomputers." *Proceedings of Scalable High Performance Computing Conference SHPCC-92*. 195 - 199. Piscataway, NJ: IEEE Press, 1992.

48. Lucasius, C.B., et al. "A Genetic Algorithm for Conformational Analysis of DNA." *Handbook of Genetic Algorithms* edited by Lawrence Davis, chapter 18, 251–281, New York: Van Nostrand Reinhold, 1991.

49. Malony, Allen D, et al. "Traceview a Trace Visualization Tool.". 19–28. September 1991.

50. Margaret Simmons, Rebecca Koskela and Ingrid Bucher, editors. *Instrumentation for Future Parallel Computing Systems*. ACM Press Frontier Series, 1989.

51. Mclaren, Russell and William A Rogers. "Instrumentation and Performance Monitoring of Distrubuted Systems." *Proceedings of the 5th Annual Distrubuted Memory Computing Conference.*. 1180–1186. Piscataway, NJ: IEEE Press, 1991.

52. Merkle, Laurence D. *Application and Parallelization of Messy Genetic Algorithms*. MS thesis, Air Force Institute of Technology, WPAFB OH 45433, December 1992.

53. Messina, Paul. "A Critique of Paragon I/O Facilities," *Proceedings of the Intel Supercomputer Users Group* (November 1993).

54. Nissen, Volker. *Evolutionary Algorithms in Management Science*. Technical Report, Universitaet Goettingen Germany, July 1993.

55. Noe, Roger J. *Pablo Instrumentation Environment User's Guide*. Department of Computer Science University of Illinois, Urbana, Illinois 61801, August 1993.

56. Open Software Foundation, Englewood Cliffs, New Jersey. *OSF/Motif^{TM} Programmer's Guide*, 1990.

57. Pankaj Mehra, Sekhar Sarukkai and Melisa Schmidt. "AIMS - An Automated Instrumentation System," *Intel Supercomputer User's Group* (1993).

58. Paul Chase, Charles Wright and others. *The AFIT Algorithm Animation Facility (Users Guide)* (4 Edition). School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB OH, February 1994.

59. Pearl, Judea. *Heuristics*. Reading MA: Addison-Wesley Publishing Company, 1989.

60. Pease, Daniel, et al. "PAWS: A Performance Evaluation Tool for Parallel Computing Systems," *IEEE Computing*, 18–29 (January 1991).

61. Raalte, Thomas Van, editor. *XView Reference Manual*. Sebastopol CA: O'Reilly & Associates, Inc, 1991.

62. Reed, Danieal A. and other. *An Overview of the Pablo Performance Analysis Environment*. Department of Computer Science, University of Illinois, Urbana, Illinois 61801, November 1992.

63. Reynolds, Jr., Paul F., "A Spectrum of Options for Parallel Simulation," 1988.

64. Reynolds, Jr., Paul F. "Comparative Analyses of Parallel Simulation Protocols." *Proceedings of the 1989 Winter Simulation Conference*. 671–679. 1989.

65. Reynolds, Jr., Paul F. and P.M. Dickens, "SPECTRUM: A Parallel Simulation Testbed," 1989.

66. Robertson, George G. "Parallel implementation of genetic algorithms in a classifier system." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 148–154. Hillsdale NJ: Lawrence Erlbaum Associates, Inc., 1987.

67. Rover, Diane T. *Visualizing the Performance of SPMD and Data-Parallel Programs*. Technical Report, Lansing, MI: Michigan State University, August 1992.

68. Rover, Diane T, et al. "Performance Visualization of SLALOM." *Proceedings of the 6th Annual Distrubuted Memory Computing Conference.*. 543-549. Piscataway, NJ: IEEE Press, 1991.

69. Rumbaugh, James. *Object-oriented Modeling and Design*. Englewood Cliffs, New Jersey 07632: Prentice-Hall, Inc., 1991.

70. Sartor, JoAnn M, et al. "Mapping Predence-Constrained Simulated Tas' s for a Parallel Environment," *Proceedings of the Sixth Distributed Memory Computing Conference* (April 1991).

71. Sarukkai, Sekhar R. and Allen D. Malony. "Perturbation Analysis of High Level Instrumentation for SPMD Programs," *ACM*, 44 – 55 (1993).

72. Sawyer, George A. "Simulation of a Ring Connection Network on 80386 Based Intel Hypercube Computers." Result of AFIT classwork.

73. Sawyer, George Allen. *Extraction and Measurement of Multi-Level Parallelism in Production Systems*. MS thesis, AFIT/GCE/ENG/90D-04, Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, December 1990 (AD-A230498).

74. Selvakumar, S and C. Siva Ram Murthy. "An effecient algorithm for mapping VLSI circuit simulation programs onto multiprocessors," *Parallel Computing North-Holland*, *17*:1009–1016 (1991).

75. SiliconGraphics Computer Systems. *IRIS Explorer User's Guide*, 1992.

76. Software, Green Hills. *iPSC/2 C Language Reference Manual* (Rev 2 Edition). Intel Corporation, Supercomputer Systems Divisision, 15201 N.W. Greebreier Parkway Beaverton, Oregon 97006, November 1988.

77. Soule, Larry and Anoop Gupta. "Characterization of Parallelism and Deadlocks in Distributed Digital Logic Simulation." *26th ACM/IEEE Design Automation Conference*. 81–86. 1989.

78. Sporrer, Christian and Herbert Bauer. "Corolla Partitioning for Distributed Logic Simulation of VLSI-Circuits." *7th Workshop on Parallel and Distributed Simulation*. Piscataway, NJ: IEEE Press, 16-19 1993.

79. Sun Microsystems, Inc. *SunView Programmer's Guide*, 1990.

80. Sun Microsystems, Inc. *SunView System Programmer's Guide*, 1990.

81. Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, October 1991.

82. Thomas Williams, Colin Kelly John and others, "GNUPLOT An Interactive Plotting Program." Distributed with Gnuplot, 1990.

83. VanHorn, Prescott J. *Development of a Protocol Useable Guideline for Conservative Parallel Simulations*. MS thesis, Air Force Institute of Technology, 1992.

84. Williams, Edward M. *Graphical Representation of Parallel Algorithmic Processes*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

85. Williams, Edward M and Gary B Lamont. "A Real-Time Parallel Algorithm Animation System." *Proceedings of the 6th Annual Distrubuted Memory Computing Conference.*. 551–561. Piscataway, NJ: IEEE Press, 1991.

86. Wright, Charles R. *X-AAARF: An X Window Based Version of the AFIT Algorithm Animation Research Facility*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Effective Parallel Algorithm Animation	5. FUNDING NUMBERS
6. AUTHOR(S) Paul W. Chase, Flight Lieutenant, RAAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GE/ENG/94M-04
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The AFIT Algorithm Animation Research Facility (AAARF) was developed by the Air Force Institute of Technology (AFIT) as a teaching aid for data structures and algorithm design. In particular, an extensive set of performance animations has been developed for the Intel iPSC Hypercube parallel processing system.

This research focuses in part on developing animation support for discrete event simulation, mission routing, and evolutionary algorithms based on abstract representations of parallel algorithm behavior. The effort also builds extensions to the AAARF system and examines direction for further research. An innovative adaptable application-specific animation construction environment has been designed and implemented. The environment provides a set of composible graphics objects and a flexible event mapping and transformation system that allows development of arbitrary animation formats. The system also reduces operational complexity via a simplified event format. The resulting visualization system is inherently portable between architectures, easily extensible to meet specific user animation requirements, and successfully deals with scalability problems associated with massively parallel processing systems.

14. SUBJECT TERMS Algorithm Animation, Visualization, Parallel Processing			15. NUMBER OF PAGES 193
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL